A DATA DICTIONARY FOR THE INGRES DATA BASE MANAGEMENT SYSTEM

by

LOREN WILSON

B. A., Kansas Wesleyan, 1980

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

Major Professor

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION AND OBJECTIVES

1.1 INTRODUCTION

A Data Dictionary(DD) is a data base software tool used primarily to hold the metadata of an organization. The metadata is what a data element is, and where it can be found. The metadata is used in one of two ways, either statically or dynamically.

A data dictionary that processes the data statically stores the metadata but isn't accessed for every transaction to the data base. The metadata is accessed only when the user specifically requests the transaction to use the data dictionary. This can lead to a situation in which a large number of data elements in the data base are not included among data definitions in the data dictionary.

A fully dynamic data dictionary is accessed for every transaction to the data dictionary. This ensures that the data base is fully dependent on the data dictionary for its metadata. Thus, the data in the data base will always be the same as the data definitions in the data dictionary.

Recently researchers have suggested using the data dictionary in the design cycle of a data base. This procedure ensures that the data in the data base is the same as the data dictionary metadata. This work discusses the partial design and implementation a data dictionary that will dynamically create a data base schema from metadata that has been input. Since the data base will be fully dependent on the data dictionary for its data, the data dictionary will be dynamic in nature.

1.2 OBJECTIVES

The objective of this work has been to partially design and implement a data dictionary that is used during the design process of a data base to dynamically create a relational data base schema for an INGRES data base management system. Due to time and resource constraints, the data dictionary is not fully dynamic. After the data base schema is created, the data dictionary is static in that it does not participate actively in transactions on the data base that have been created. This work describes dynamically creating a data base schema from metadata in the data dictionary. The thesis lays a foundation that can be used as the basis for a fully dynamic data dictionary.

The data dictionary stores metadata for a data base. However, it does not store information about where data elements are, so it can not be construed as a data directory. The metadata is interactively input by a user, and dynamically checked for data redundancy as it is input. The user has the option of viewing the metadata in the data dictionary after it has been input. After the metadata is entered, a data base schema may be dynamically created from the metadata that has been input into the data dictionary.

The problem was broken down into four areas:

1) Development of a data dictionary data base to store the metadata for data base elements in the INGRES DBMS.

2) Development of interactive input programs.

3) Development of interactive output programs.

4) Davelopment of tha techniques and programs to dynamically create a relational data base schema.

1.3 THESIS GUIDE

Chapter Two discusses tha functions that an average data dictionary can be expected to handle without difficulty. Further, the second chapter discusses the differences between the two typas of data dictionaries, static and dynamic.

Chapter Threa discusses the development of the data dictionary data base for the INGRES data base management system, and the interactive input and output programs developed.

Chaptar Four discusses tha techniquas that ara used to create a data base dynamically from metadata residant in the data dictionary. A discussion of why the data dictionary is used in tha data base dasign process is also includad.

Chapter Five discusses tha actual implementation of the data dictionary. The discussion covers tha data dictionary, explaining what each of the modules does.

Chapter Six summarizes this work and discusses further work that can be done with the data dictionary in order to further enhance its dynamic charactaristics.

CHAPTER TWO

DATA DICTIONARY BACKGROUND

2.1 DATA AS A RESOURCE

The concept of "data as a resource" is a recent idea [Leong-Hong 82]. Organizations have always valued their primary resources, i.e., personnel, money, and materials. Historically, data has not been considered a resource primarily because it lacks two characteristics of these primary resources, namely, allocability and scarcity [Leong-Hong]. However, although data does not possess either of these characteristics, the ability to collect, store, and verify the integrity of their data is a high priority with organizations. Recognition of the importance of data has given rise to the idea that data is an important resource.

To help in the storage and management of data, many organizations have purchased a Data Base Management System (DBMS). A DBMS allows the data to be stored in a central location that is accessible to any user in the organization. However, there is still a major problem with the use of a DBMS. Although a DBMS allows the data to be stored, it doesn't provide the capability to describe the data.

Data dictionaries were developed to store the definitions of the data contained in a DBMS. With a data dictionary, organizations have the capability to store the metadata of their organization. Thus the organization gains control over its data and can verify the integrity of the data.

2.2 FUNCTIONS OF A DATA DICTIONARY

The functions of a general use data dictionary will be discussed in this section.

2.2.1 METADATA DEFINITION

The primary function of a Data Dictionary(DD) is to store the metadata of an organization. This metadata allows the organization to gain control over its data resources by collecting, storing, and verifying the integrity of the data elements in the organization. Unfortunately, no standard exists for what should be stored as metadata [Van Duyn 82]. The same general metadata is stored in most data dictionaries, but the scope, size, and complexity of the DD depends on the type and size of the organization for which the data dictionary is designed and the commercial data dictionary used [Van Duyn 82].

For purposes of discussion, a hypothetical data dictionary is presented in Figure 2.1. This data dictionary is an example of what a reasonably good data dictionary may store for each data element. It is not meant to be a paradigm or standard. There are not any standards that currently exist for metadata definition. If properly implemented, this format allows the DD to perform the functions discussed in this chapter.

The DATA ELEMENT NAME is the unique name given to the data element. This name is the name used in the system wherever the data element is accessed. Using another name (a SYNONYM) instead leads to data redundancy.

The DATA ELEMENT NUMBER is a unique number assigned to the data element. This number is primarily for fast access to the

DATA ELEMENT NAME	: unique name of the data element.
DATA ELEMENT NUMBER	: unique number assigned to the data element for easy access.
SYNONYM(S)	: any other name that the data element is known by in the data base.
TYPE	: the format of the data element value. i.e., character, numeric, or alphanumeric.
LENGTH	: length or maximum length of the data element.
ORIGINATING SYSTEM	: system where the data element was first defined.
SOURCE	: department that first generated the data element.
FILES	: files that contain the data element.
REPORTS	: reports that have the data element contained in them.
FORMS	: forms that use the data element.
DEFINITION	: detailed description of the data element.

Figure 2.1: SAMPLE DATA DICTIONARY METADATA

data element. It is not used in programs or reports.

The SYNONYM is any other name by which the data element may be known. A synonym should not be widely used in the system. In fact it should hardly be used at all since it leads to a data redundancy.

The TYPE of the data element states what the format of the data element should be. This field usually contains one of the following: "character," which signifies word or string; "numeric," which signifies an integer or decimal; or "alphanumeric," which signifies a mix of characters and numerals. An example of the latter is a street address such as "1455 Water Street."

The LENGTH of the data element varies with the TYPE of data element. If the type is character or alphanumeric, the length is the maximum number of characters allowed in the word or string. If the type is numeric, the length is the maximum integer or decimal allowed.

The ORIGINATING SYSTEM is the name of the system or application where the data element is first defined. An example of originating systems is a certain data base or a particular application program.

The SOURCE is the name of the division, department, or shop where the data element is first generated. An example of a source is the accounting division or the personnel department.

The FILES field is a list of the system's files that have the data element residing in them. These are files from one or more of the data bases in the system.

The REPORTS field is a list of the reports that display the data element. Examples of reports are invoices, statements, and annual financial reports.

The FORMS field is a list of the organization's forms that contain the data element. Examples are job applications and purchase orders.

The DEFINITION is a detailed description of the data element. This field can be used in the detection of data redundancy.

2.2.2 MINIMIZATION OF DATA REDUNDANCY

Data redundancy is a major design consideration problem in any data base. Limitation of data redundancy minimizes storage usage and integrity maintenance problems [Van Duyn 84]. Figure 2.2 illustrates the various forms of data redundancy.

Strict adherence to the metadata standards described in the data dictionary helps eliminate data redundancy problems. The adherence to standards simplifies the enforcement of standard usage and consistency in documentation for data elements [Leong-Hong 77].

REFERENCE REDUNDANCY:
A single data element has several different names within the system.

FORMAT REDUNDANCY:
Variation of the type and length of a data element.

GROUP REDUNDANCY:
A data element is created to reference one or more data elements. This adds unneeded data elements to the system.

OCCURRENCE REDUNDANCY:
Repetitious names identify multiple generations of the same data element. This adds to the complexity of the data definition.

DEFINITION REDUNDANCY:
A single data element is used for more than one purpose within the system. It is the worst of the data redundancies because it can cause the whole system to be more complex allowing for the definition redundancies.

STORAGE REDUNDANCY:
A single data element is stored in more than one location. [Durrell 83]

Figure 2.2: DATA REDUNDANCIES

2.2.3 DATA AND RELATIONSHIP HANDLING

The ability to track and update the data helps maintain data usage integrity and efficiency. Tracking of the data is the process of following the data from its first definition and generation, to the current reports and forms that are using the data. The ability to track the data is accomplished by the strict use of the data constraints outlined by the data dictionary. Users throughout an organization can discover where a data element is stored, where it is being used, and how it is being used [Van Duyn 82].

The data dictionary has the ability both to store the relationships between data elements within one data base, and to store the relationships between data elements in different data bases that reside in the same system [Van Duyn 82].

The data and relationship handling functions can be useful in the development and maintenance cycle of a system [Durrell 83]. The ability to handle both data and relationships allows the data dictionary to participate in system changes, provide reliable documentation, and enforce an adherence to standards. The above information can be useful to both system programmers and data administrators [Van Duyn 82].

The primary advantage of a data dictionary in data and relationship handling is the control an organization gains over its data. The management of the organization can count on the data dictionary to provide accurate information about the data in the organization. This information can aid in the decision making process of the organization [Van Duyn 82].

2.2.4 SECURITY

The data dictionary can be used as an additional security level into the system. It can protect access to information about the organization's resources and access to dictionary functions. These capabilities can be accomplished in one of the following ways.

1) Maintenance of a user profile for each user or group of users. This profile can include user ID, password, security level, application files the user is allowed to access, records the user is allowed to access, the terminals the user is allowed to use, and the group to which the user belongs.

2) Dynamic allocation of time-limited passwords to eliminate the problem of human error in the allocation of passwords.

3) Maintenance of a log file of every access to sensitive files and programs [Van Duyn 84].

2.3 TYPES OF DATA DICTIONARIES

This section contains a discussion of both static data dictionaries and dynamic data dictionaries. Not all data dictionaries fall neatly into one of these two categories. The majority of them fall on a wide spectrum between the two end points. Commercial data dictionaries that are introduced as either static or dynamic are in actuality either generally static or generally dynamic. DBMS-dependent data dictionaries as compared to stand-alone data dictionaries will also be discussed.

2.3.1 STATIC DATA DICTIONARY

A static data dictionary, also known as a passive DD, is the older of the two types. A static data dictionary does not

require direct interaction with language compilers, the DBMS, or the operating system [Ross 81]. It can provide information about the data, but does not participate actively in the handling of transactions [Martin 83]. A diagram of a static data dictionary can be seen in Figure 2.3. The diagram illustrates that the DBMS can process the application software programs without accessing the data dictionary.

The primary attribute of a static data dictionary is its lack of integration with the rest of the other software elements in the system. An example is the IBM DB/DC data dictionary and its related software. The DB/DC stands apart from the rest of the system. The system stores data definitions in at least six places. The data in each of these places may agree, but it is not required to by the system. The places where data definitions

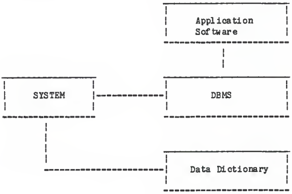

Figure 2.3: STATIC DATA DICTIONARY

can be stored with this system are:

1) The DB/DC dictionary.
2) The DBD/PSB libraries.
3) The COBOL copy library.
4) The Data Base Design Aid.
5) The GIS data definition tables.
6) The Application Development Facility(ADF).
[Curtice 81]

Characteristics of a static DD are:

1) Data base schemas are stored but not used at compile time. Therefore, the data base is not dependent on the data dictionary for its metadata.

2) Data base subschemas are stored but not always accessed to run the subschema.

3) File descriptions are kept for each application program. However, a program is not required to use that information [Ross 81].

The major disadvantage of a static data dictionary is even though the capabilities of a data dictionary are present, these capabilities are not used to their fullest extent. None of the processes in the system are required to access the data dictionary for metadata.

The advantage of a static DD lies in the fact that the

DB/DC Data Dictionary	-	IBM
Datamanager	-	MSP Inc.
UCC-10	-	University Computing Co.
Data Catalogue 2	-	Synergetics Corp.

Figure 2.4: COMMERCIAL STATIC DATA DICTIONARIES

capability of the data dictionary does exist. However, for an organization to use its data dictionary to its fullest capabilities, it must set up an internal system that anaures that all transactions usa tha data dictionary.

Several commercial static data dictionaries ara available. (See Figure 2.4) A commercial data dictionary is similar to a commercial data base management system in that it is purchased from a vandor and then implemented to fit the particular organization.

2.3.2 DYNAMIC DATA DICTIONARY

A dynamic data dictionary, somatimes called active, is used to actively control the use of the metadata and tha data base anvironmant. It rasides in tha mainstream of the data base pro-

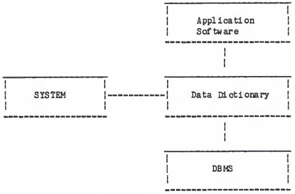

Figure 2.5: DYNAMIC DATA DICTIONARY

cessing activities and is a control mechanism for the processing functions [Davis 84]. These processing functions include query languages, report generators, and application development aids. A DD is considered dynamic if a program or process is fully dependent on the data dictionary for its metadata [Leong-Hong 82]. A diagram of a dynamic data dictionary is in Figure 2.5. A list of generally dynamic commercial data dictionaries is in Figure 2.6.

The primary advantage of a dynamic data dictionary is its absolute control over the data resources. In a dynamic environment, all processing goes through the data dictionary. Thus, the DD is used to its fullest capabilities at all times.

Other benefits exist in a dynamic data dictionary environment as well. One additional benefit is that control over the metadata usage is enhanced. Since all components of the system are dependent on the DD for their metadata, the whole system can be controlled from the DD. Any component that is not authorized can be blocked by the DD's withholding the metadata. The DD controls any changes to the metadata. Therefore, another benefit of a dynamic DD is that any changes to the metadata are reflected

Data Dictionary	-	Applied Data Research
Control 2000	-	Intel
Integrated Data Dictionary	-	Cullinet
Data Control System	-	Cincom
Predict	-	Software AG

Figure 2.6: COMMERCIAL DYNAMIC DATA DICTIONARIES

throughout the entire system.

The primary disadvantage associated with a dynamic data dictionary is the overhead introduced to store a system's metadata [Allen 82]. A central repository of metadata is cumbersome and can cause bottlenecks if several system components are trying to access it concurrently.

2.3.3 DBMS-DEPENDENT DATA DICTIONARIES

Several of the commercially available data dictionaries are DBMS-dependent. A DBMS-dependent DD is designed for use with a particular data base management system. These systems are usually sold by a particular vendor to integrate with their own general purpose DBMS. Examples of DBMS-dependent data dictionaries are the DB/DC Dictionary from IBM, which works with the IMS data base management system; and the Integrated Data Dictionary from Cullinet, which works with the IDMS DBMS [Leong-Hong 82].

A dependent data dictionary can contribute to a system's optimization. Since the DD is tightly connected to the single DBMS, it can easily generate control blocks and supply a comprehensive inventory of DBMS and non-DBMS files [Van Duyn 82].

The main problem with a dependent DD lies with its limitation of working with only a particular DBMS. If an organization changes DBMS's or adds a second DBMS, a new DD would need to be purchased and the new data dictionary would need to be implemented along with the new DBMS. And whenever a second DD is added, duplicate data definitions can be created if proper precautions are not followed [Marti 84].

2.3.4 STAND-ALONE DATA DICTIONARY

A Stand-alone data dictionary does not need a particular DBMS to operate; such dictionaries are designed to run with different types of DBMS's. In extreme cases, a stand-alone data dictionary can operate with only a normal flat file system.

An example of a stand-alone data dictionary is the Datamanager from MSP, Inc. Datamanager is known for its wide range of metadata generation capabilities and its ability to support five of the major DBMS's [Leong- Hong 82]. The DBMS's it can support are ADABAS, IDMS, IMS, MARK IV, AND TOTAL.

A stand-alone data dictionary has the capability to edit and verify all data entities before storing them, thus ensuring consistency in all data definitions [Van Duyn 82]. Through separate DBMS interfaces, it can generate control blocks and supply a comprehensive inventory of DBMS and non-DBMS files.

The primary disadvantage with a stand-alone data dictionary is the overhead it adds to the system. For a single DBMS environment, a stand-alone DD can add unneeded overhead and complexity to the system.

CHAPTER THREE

DESIGN OF THE DATA DICTIONARY

3.1 DATA DICTIONARY COMPONENTS

The following components are used in the design of the data dictionary.

1) A data dictionary data base. The data dictionary data base contains descriptions of the data, format of the data, and other names for the data (the metadata).

2) Programs to enter metadata. These programs are interactive with the user and allow a user to add, modify, or delete metadata from the data dictionary data base.

3) A program to retrieve data from the data dictionary data base. This program is interactive.

4) Interface to the data base management system. This allows the data dictionary to create a data base schema in the DBMS.

Steps one through three are addressed in this chapter and step four in chapter four.

3.2 TEST OF DATA DICTIONARY FUNCTIONS

A comparison of the components stated above with the functions discussed in Chapter Two provides the following analysis.

Metadata definition

This function is addressed in component one. A Data Dictionary Data Base (DDDB) has be developed that stores the data elements and their definitions (the metadata).

Minimization of data redundancy

As the metadata is interactively entered using component

two, the data dictionary dynamically checks all new data elements for redundancy.

Data and relationship handling

In component four, the data dictionary handles both the data and relationships. Since the data dictionary creates the data base it knows what the data and relationships are at creation time, but does not handle any tracking once the data base is operational.

Security

This is the only function that is not handled by any of the components. Component two or component four could be expanded to include this function.

3.3 THE DATA DICTIONARY DATA BASE

3.3.1 SELECTION OF A DBMS

The INGRES data base management system has been chosen to implement the data dictionary. INGRES is a relational DBMS that is becoming more popular in industry. INGRES is becoming widely used on mini computers, and has a powerful query language called "QUEL" that is easy to use. The QUEL language can be embedded into a C language program and run through a separate pre-processor called "EQUEL" which allows the program to interface directly with the data base.

3.3.2 FORM OF THE DATA DICTIONARY ELEMENTS

A minimal metadata description area is allocated for the data dictionary. A large data dictionary could have easily been

created (as in Figure 2.1), but it would have made the implementation of the data dictionary a much more lengthy process.

The data dictionary data base schema that is used can be seen in Figure 3.1. There are three separate entities in this model. The main entity is "Element,", which is related to the "Synonym" and "Instance" entities by 1-N relationships. That is, for a single element entity, there can be several "Synonym" entities, and also several "Instance" entities. See Figure 3.2 for a description of the fields in each entity.

The first entity to be examined is the Element entity. The first field in Element is the NAME, which is the name of the data element that is being described. It is of type character, and can not be over 20 characters in length. The DESCRIPTION field is a detailed description of the data element. The TYPE field describes what the type of the data element will be. This field is filled with either "character," "integer," "floating" (stand-

```
 --------------                          --------------
|              |  1                   N |              |
|   ELEMENT    |------------------------|   SYNONYM    |
|              |                        |              |
 --------------                          --------------
       |1
       |
       |
       |N
 --------------
|              |
|   INSTANCE   |
|              |
 --------------
```

Figure 3.1: DATA DICTIONARY SCHEMA

ing for floating decimal), or "alphanumeric." The LENGTH field is the length of the data element. If the TYPE is either character of alphanumeric, then the LENGTH is the maximum allowed number of characters. If the TYPE is integer or floating, then the LENGTH is the maximum number of bytes needed to store the data element. The ENTITY_NAME field is either "Y" or "N," signifying whether the data element is the name of an entity in the data base. This information is not required (and therefore was not included in the metadata definitions in Chapter One), but it is included here to aid in the definition of the data dictionary.

The Synonym entity lists the aliases that the data element has. A separate entity has been made for Synonym so that more than one alias can be listed for each data element. The Synonym entity also aids in the detection of redundant data elements. The separation makes it easier to access the different aliases

Element

NAME	-	Character 20
DESCRIPTION	-	Character 30
TYPE	-	Character 12
LENGTH	-	Integer 2
ENTITY_NAME	-	Character 1

Synonym

NAME	-	Character 20
ALIAS	-	Character 15

Instance

NAME	-	Character 20
KEYWORD	-	Character 1
ENTITY	-	Character 20

Figure 3.2: DATA DICTIONARY ENTITIES

for each data element. The NAME field in the Synonym entity is exactly the same as the NAME field in the Element entity. The NAME field sets up the relationship between the Synonym and Element entities as is needed in a relational DBMS. The ALIAS field is the name of the alias.

The Instance entity is a list of the separate instances of the data element in the data base and is related to the Element entity by the NAME field as in the Synonym entity. The NAME field is exactly the same as the NAME field in the Element entity. The KEYWORD field is a one character field, either "Y" or "N." The single character signifies whether the instance of this data element is a keyword in a data base entity. The ENTITY field signifies the name of the entity in the data base to which that data element instance belongs.

Returning to the Element entity, if there is a "Y" in the ENTITY_NAME field, that data element has no corresponding instances. A "Y" signifies that the data element is the name of a data base entity and can not have any other instances in the data base. If this were not the case, there would be a definition redundancy within the data base.

The inclusion of the Instance entity in the data dictionary is not standard. It is included here, however, to document where in a data base a particular data element is used. The Instance entity aids in the implementation of the data base creation capability. (further explanation provided in Chapter Four)

3.3.3 EXAMPLE DATA DICTIONARY DATA BASE

An example of a data dictionary built from two data base

Data Base Entities

Client	Employee
Name	Name
Age	Salary

Data Dictionary Entities

Elements

Client	Name	Employee
Client	Name	Employee
Buyer of services	Name of a person	Company worker
Character	Character	Character
20	20	20
Y	N	Y

Salary	Age
Salary	Age
What employee is paid	# of years old
alphanumeric	integer
11	99
N	N

Synonyms

Client	Name	Name	Employee
Client	Name	Name	Employee
Buyer	person name	last name	slave

Employee	Salary	Salary	Age
Employee	Salary	Salary	Age
worker	pay	earnings	years old

Instances

Name	Name	Salary	Age
Name	Name	Salary	Age
Y	Y	N	N
Client	Employee	Employee	Client

Figure 3.3: EXAMPLE DATA BASE AND DATA DICTIONARY

entities can be seen in Figure 3.3. The two data base entities

are minimal. They represent a small Client entity that stores the client's name and age, and a small Employee entity that stores the employee's name and salary. The data base contains the data, and the data dictionary contains the definitions of the data (the metadata). The metadata outlines the form, structure, and semantics of the data elements.

There are five different data elements from the data base entities. A corresponding tuple in the Element entity for each of them is entered in the data dictionary. This example describes at least one Synonym entity for each data element. However, Synonym entities are not required by the data dictionary. The DD described here does not have a limit on the number of aliases that a data element can declare, though some commercial data dictionaries have such a restriction.

Four Instance entities exist in this example. Neither the Client nor Employee data elements have an instance entity. Both Client and Employee are the names of an entity and neither has any instances declared. If either were allowed to have any other instances within the data base, a definition redundancy would result. Of the Instance entities that exist, two are from the Name data element. One of these corresponds with the Client and the other with the Employee. Age and Salary each have only one Instance entity, because they appear only once in the data base.

Assume that the following entity has been added to the data base:

<u>Manager</u>
Name
Salary

The Manager data element would first be added to the data dictionary. Synonym entities would also be added. There would not be any new Instance entities for "Manager" since it is the name of an entity.

When the Name and Salary data elements are added to the data dictionary, it is discovered that they already reside in the DD. The only thing that needs to be inserted into the dictionary is a new Instance entity for each data element. These entities would look like:

<u>Name</u>	<u>Salary</u>
Name	Salary
Y	N
Manager	Manager

Whenever new instances of a data element are added to the data dictionary, the only thing that is added is a new Instance entity. The rest of the data element is left unchanged.

<u>3.4 INPUT INTERACTION</u>

The only alternative to interactive input programs for the data dictionary is to require the user to manually access INGRES to enter metadata. Manual access causes an incredible slowdown of the processing time and this greatly hampers the effectiveness of the data dictionary. To enter metadata manually requires the user to access the Element entity and enter each data element, then access the Synonym entity and enter the same data element name for each new alias. In other words, the user inputs the same data element name several times. An interactive input program has been designed to speed up the processing time.

Ease of use to add, modify, or delete a data element from the data dictionary was the principal goal in the design of the input program. Simplicity is echieved by having the user use a single virtuel entity instead of the three physical entities. The virtual entity is implemented in the interactive program, simplifying the process of interacting with the DD, and allowing the user to menipulete the data dictionary more rapidly.

A secondary goal was to eutomate redundancy checking capabilities. Redundency checks are dynamically performed as the metedate is input into the data dictionary. When e new data element is edded to the DD or an existing one is modified, the data dictionary checks it against other elements that already exist. The main data redundancy checked for is definitional redundancy (e single data element used for more than one purpose), but other kinds of redundencies that can be detected are format and storage. (See Figure 2.2 for the different types of redundancies)

The data dictionary makes the first comparison as soon as it receives the Name field of a data element. Name will be checked for redundancy before the user is allowed to continue the input process. The data dictionary compares the Name to existing data element Names, and if it matches eny existing data element Name, an error message will be output and the new data element is not permitted in the system. If the new Name does not match any of the existing data element names, the dictionary will make a second comparison, this time to the to the existing aliases. If the new name does not match any of the aliases, the data element is conditionally added to the DD. Otherwise, the new data element

is disallowed.

The next metadata item entered is the description of the data element. Ideally, the description would be semantically compared to the other data element descriptions. Semantic comparison allows the data dictionary to compare the actual meaning of descriptions. But semantic comparison is not possible at this time. To compare the descriptions word for word is fruitless; two descriptions can mean the same thing and yet be worded completely differently. Thus after consideration, this form of redundancy checking has been omitted from the design.

The data dictionary next makes a comparison when an alias is entered into the system. This comparison is a two step process. The data dictionary first compares the new alias to the existing data element names. A match disallows the new data element because of redundancy. A point should be made here about data bases. Data bases are complex to design and manipulate. Few data base rules are "written in stone"; exceptions seem to abound. Such is the case here. The possibility exists that a data element can have an alias that is the same as another data element name, yet be semantically different. If this situation occurs, the data dictionary will query the user to see if the data element should be kept. If the user declares the data element redundant, it is removed from the data dictionary. The second comparison will be to compare the new alias to all existing aliases. If the alias does not match, the only way a data element can then be removed from the data dictionary is for the user to specifically command the removal.

3.5 SAMPLE INPUT

At this point, example data elements will be added to the data dictionary (Figure 3.3) to illustrate the reaction of the data dictionary. The focus is on what data elements are allowed to become part of the data dictionary along with explanations of why rejected data elements aren't accepted. The data base describes a simple car dealership.

Suppose the first data element added to the system is "Name." "Name" is the name of a car (i.e., Buick, Ford). The new data element looks like:

```
               Name
NAME         : Name
DESCRIPTION: model of the car
TYPE         : Character
LENGTH       : 15
ENTITY       : N
```

The data dictionary takes the new data element name "Name" and makes the first comparison. "Name" is disallowed because it matches with "Name," the "Name of a person" which is already in the system. The user interactively changes it to "Model" in which case it is accepted. The new data element then looks like:

Model
Model
Kind of car
Character
15
N

The next data element entered is "Pay":

Pay
Pay
Salary of Employees
alphanumeric
11
N

The data dictionary takes "Pay" and makes the first set of comparisons. "Pay" is compared to the existing data element names with no matches. Next "Pay" is compared to the existing aliases. This time there is a match. "Pay" is the name of an alias for the data element "Salary," so the new data element "Pay" is disallowed. The user decides to use "Salary" instead of "Pay."

The data element "Customer" is added to the data dictionary:

Customer
Customer
Prospective car buyer
Character
20
N

The data dictionary compares "Customer" to the existing data element names. There are no matches, so it compares "Customer" to the existing data element aliases. Again there are no matches.

The data element is conditionally added to the system. An alias is then added to the data dictionary for Customer:

Customer
Customer
Purchaser

The data dictionary takes the alias "Purchaser" and compares it to the existing data element names. There are no matches, so it compares "Purchaser" to the existing aliases. Again there are no matches. The alias "Purchaser" is conditionally accepted for the data element "Customer." Another alias is then added for Customer:

Customer
Customer
Client

The data dictionary takes the alias "Client" and compares it to the data element names. A match is made with the data element name "Client" which is a "Buyer of services." The data dictionary issues a query on whether to keep "Customer." The user decides "Customer" is a redundant data element, so the entire Customer data element, even the parts of it that were conditionally accepted to the system, are removed.

3.6 OUTPUT INTERACTION

The interactive output is designed for the user to view one or all the data elements grouped by data element name. The user can view the entities as if they are in a single file. A sample

user view of a data element residing in the data dictionary is shown in Figure 3.4.

```
This is what Name element looks like now.

NAME =            Name
DESCRIPTION =     Buyer of services
TYPE =            Character
LENGTH =          20
ENTITY_NAME =     N
Alias =           person name
Alias =           last name
INSTANCE 1
   KEYWORD =         Y
   ENTITY =          Client
INSTANCE 2
   KEYWORD =         Y
   ENTITY =          Employee
```

Figure 3.4: SAMPLE OUTPUT FOR A DATA ELEMENT

CHAPTER FOUR

DYNAMIC DESIGN

4.1 DYNAMIC DESIGN PROBLEMS

Most of the data dictionaries in use today have been purchased from vendors. These commercial data dictionaries are mostly static in nature. The vendors for these data dictionaries are wary of letting out detailed technical information on their packages. Few journal articles or books about the design and implementation of a data dictionary are available; most articles describe a taxonomy of data dictionaries or describe how to decide which of the available commercial packages to purchase. Because of the dearth of information about the design of dynamic data dictionaries, the techniques used in this work to make the DD dynamic at compile time were developed without information on the implementation level.

This chapter outlines the techniques used to dynamically create a data base from the metadata resident in the data dictionary. These techniques ensure that the data base is fully dependent on the data dictionary for its metadata. This makes the data dictionary dynamic in nature at data base compile time.

4.2 DATA DICTIONARY USE PROBLEMS

Data dictionaries are used mostly as a documentation aid for data base management systems [Leong-Hong 82]. This concept relates directly to static data dictionaries. A static DD contains the documentation for the DBMS, but is rarely used for any type of processing. A data dictionary in this type of environment is purchased after a data base is operating. The metadata

from the data base is entered into the data dictionary and then tested for redundancies. The integration of a data dictionary into a system at this phase in a data base life cycle can be a long and difficult process. Some of the problems include changing data bases that are already running, and standardizing names from different files in the data base [Wearing 73].

To integrate a fully dynamic data dictionary is a difficult process. Most organizations have thousands of application programs, reports, and files that need to be captured by the data dictionary [Allen 82]. A "convert" function can be used to populate the data dictionary, but there can still be data redundancy in the data dictionary [Allen 82]. The ideal would be to integrate the fully dynamic data dictionary into the system at set up time [Leong-Hong 82].

Along with integrating a dynamic data dictionary into the system at set up time, some researchers suggest that the data dictionary should be used in the actual design of a data base [Marti 84][Leong-Hong 82]. Use in designing the data base would assure a single authoritative source of data definitions so the new data base is free from redundancy; and common definitions and interpretations are enforced in the entire system development [Leong-Hong 82].

The data dictionary described in this research has been designed specifically for the purpose of creating a data base from the metadata that resides in the DD. The use of the data dictionary in this manner ensures that the new data base is fully dependent on the data dictionary for its metadata. There is no

way for the data base to gather metadata except through the data dictionary. Other system functions can be built using the metadata in the data dictionary. When the entire system is completed, it is entirely dependent on the data dictionary for its metadata.

The data dictionary does not take any set of metadata and create a valid data base. The data dictionary should be used as a step in the design life cycle of the data base. It is highly imperative that the designers already know what the entities are and what the fields will look like in each entity before entering them into the data dictionary. The data dictionary then performs redundancy checks on the metadata and dynamically creates the data base schema from the metadata.

4.3 DYNAMIC CREATION OF A DATA BASE

The creation of a data base from a data dictionary is a lengthy process. The data dictionary goes through the following steps to dynamically create a data base:

STEP ONE

The data dictionary looks for all of the data elements that have "Y" in the ENTITY_NAME field. The "Y" signifies that the data element is the actual name of a data base entity. The DD places these data elements into a buffer.

STEP TWO

The DD takes one of the data base entity names and searches for all corresponding instances. These are found by looking at the ENTITY field in the instance entities. If the ENTITY field in the instance entity matches up with a data base entity name, then the data element name from that instance is stored in the buffer with the data base entity name.

STEP THREE

The DD determines which of the fields in an entity is the keyword(s). This is done immediately after a data element has been stored as belonging to an entity name. While still in the instance entity, the DD checks the KEYWORD field. If it is "Y," then the data element is a keyword and is flagged in the buffer.

STEP FOUR

The DD takes one of the entity names and enters it into INGRES as an entity. Each data element entity that is used as a field in the data base entity is examined. The type and length of the data element is accessed from the DD and used to create the fields in the data base.

This process dynamically creates a relational data base schema in INGRES from the data dictionary.

4.4 CREATING A SAMPLE DATA BASE

This section illustrates the process of taking a data base design that is still in the development phase, and entering the metadata of the designed data base into the data dictionary. The data dictionary system uses the data definitions to create a logical data base schema in INGRES.

4.4.1 ER DIAGRAM

An ER (Entity-Relationship) diagram is used to illustrate the sample data base. The ER diagram is a form used in the design of data bases to describe the entities of interest and the relationships between them. The entities are shown as boxes and entity attributes surround them in circles. The relationships between entities are signified by the line between the entities with the lines labeled to differentiate them. The relationships are labeled with either: "1-1" which stands for one to one; "1-N" one to many; or "N-M" many to many, which describes the number of

```
 -------------------                             ----------------------
|                   |                           |                      |
| CLIENT :          |                           | CARS:                |
|    Name           |1       CAR              N |      Name            |
|    Address        |---------------------------|      ID_Number       |
|    Credit         |      BOUGHT               |      Price           |
 -------------------                             ----------------------
           |N
           |
   BOUGHT  | FROM
           |
           |1
 --------------------
|                    |
| SALESMAN:          |
|      Name          |
|      ID_Number     |
|      Address       |
|      Salary        |
 --------------------
```

Figure 4.1: EXAMPLE ER DIAGRAM

records of each type which are related in a given instance. The ER diagram is considered a network model of a data base.

A sample data base schema represented in the form of an ER diagram is given in Figure 4.1. This data base is for a car dealership. There are three entities; Client, Cars, and Salesman. The Client is the person buying a car. Client has three attributes; Name, Address, and Credit Rating. The Client is related to the Cars entity by a 1-N relationship called Car Bought. This means that one client may buy zero, one, or many cars from the dealership. The Cars entity has three attributes, Name, ID Number, and Price. The last entity is the Salesman entity. It has four attributes; Name, Address, ID Number, and Salary. The Salesman is related to Client by a relationship

called Bought From. This is a 1-N relationship where one Salesman can sell cars to many clients.

4.4.2 INPUT INTO THE DATA DICTIONARY

When the ER diagram is finalized, the information from the ER is input into the data dictionary. The input into the DD is done by selecting an E-R entity by using the adjacent relationships and entering all of the metadata elements about the entity into the data dictionary. If one of the data elements in a subsequent entity is flagged as redundant, the data dictionary analyzes the new data element and determines if it is already in the data dictionary. If it is indeed already in the data dictionary, the only thing added to the data dictionary is a new instance for the data element. If it is decided that the new data element is different from the data element already existing in the data dictionary, a human must decide whether to change one of the data elements.

The data elements from the CLIENT entity are input first. There are not any problems with redundancy since it is the first entity entered. (The metadata for CLIENT can be seen in Figure 4.2.) The next data elements input are from the SALESMAN entity. The first element is the "Name" field which is immediately flagged as being redundant. At this point the old "Name" and the new "Name" data elements are examined. They appear to be the same data element. The only thing that is done is to add an instance for SALESMAN in the "Name" data element file. The "ID_Number" is input without any redundancy problems. However, the Address is flagged as being redundant. Once again, it is

```
NAME =          CLIENT             NAME =          Name
DESCRIPTION =   Buyer of           DESCRIPTION =   Name of a
                services                           person
TYPE =          Character          TYPE =          Character
LENGTH =        20                 LENGTH =        20
ENTITY_NAME =   Y                  ENTITY_NAME =   N
ALIAS =         Buyer              ALIAS =         Moniker
ALIAS =         Customer           ALIAS =         Last name
                                   ALIAS =         First name
                                   INSTANCE 1
                                      KEYWORD =       Y
                                      ENTITY =        CLIENT

NAME =          Address            NAME =          Credit
DESCRIPTION =   Place of           DESCRIPTION =   How much money
                residence                          they have
TYPE =          Alphanumeric       TYPE =          Alphanumeric
LENGTH =        40                 LENGTH =        10
ENTITY_NAME =   N                  ENTITY_NAME =   N
ALIAS =         City               ALIAS =         Credit rating
ALIAS =         State              INSTANCE 1
INSTANCE 1                            KEYWORD =       N
   KEYWORD =       N                  ENTITY =        CLIENT
   ENTITY =        CLIENT
```

Figure 4.2: DATA ELEMENTS FROM CLIENT ENTITY

decided that the same data element is already in the data dictionary, so an instance is added only for "ID_number." The metadata from the SALESMAN entity can be seen in Figure 4.3.

The CARS entity is entered next. The metadata for the CARS entity can be seen in Figure 4.4. When the "Name" data element is input, it is flagged as redundant. When the two "Name" data elements are compared, they are found to be completely different. It is decided by a human to change the new data element to Model. The redundancy is fixed without any major data base design problems. When the next data element "ID_Number" is input, it too is

```
NAME         = SALESMAN
DESCRIPTION  = Seller of
               cars
TYPE         = Character
LENGTH       = 20
ENTITY_NAME  = Y
ALIAS        = Seller
ALIAS        = Employee
ALIAS        = Slave
```

```
NAME         = Name
DESCRIPTION  = Name of a
               person
TYPE         = Character
LENGTH       = 20
ENTITY_NAME  = N
ALIAS        = Moniker
ALIAS        = Last name
ALIAS        = First name
INSTANCE 1
   KEYWORD     = Y
   ENTITY      = CLIENT
INSTANCE 2
   KEYWORD     = Y
   ENTITY      = SALESMAN
```

```
NAME         = Salary
DESCRIPTION  = What employee
               is paid
TYPE         = Alphanumeric
LENGTH       = 11
ENTITY_NAME  = N
ALIAS        = Earnings
ALIAS        = Pay
INSTANCE 1
   KEYWORD     = N
   ENTITY      = SALESMAN
```

```
NAME         = ID_Number
DESCRIPTION  = Unique identifi-
               cation number
TYPE         = Integer
LENGTH       = 2
ENTITY_NAME  = N
ALIAS        = SSN
INSTANCE 1
   KEYWORD     = N
   ENTITY      = SALESMAN
```

```
NAME         = Address
DESCRIPTION  = Place of
               residence
TYPE         = Alphanumeric
LENGTH       = 40
ENTITY_NAME  = N
ALIAS        = City
ALIAS        = State
INSTANCE 1
   KEYWORD     = N
   ENTITY      = CLIENT
INSTANCE 2
   KEYWORD     = N
   ENTITY      = SALESMAN
```

Figure 4.3: DATA ELEMENTS FROM SALESMAN ENTITY

```
NAME         = CARS                 NAME         = Model
DESCRIPTION  = Product being        DESCRIPTION  = Kind of car
               sold                 TYPE         = Character
TYPE         = Character            LENGTH       = 20
LENGTH       = 20                   ENTITY_NAME  = N
ENTITY_NAME  = Y                    ALIAS        = Make
ALIAS        = Automobile           ALIAS        = Brand
ALIAS        = Lemon                INSTANCE 1
                                       KEYWORD      = N
                                       ENTITY       = CARS

NAME         = Serial_Num           NAME         = Price
DESCRIPTION  = Unique factory       DESCRIPTION  = Cost of the
               given number                        car
TYPE         = Alphanumerio         TYPE         = Decimal
LENGTH       = 15                   LENGTH       = 4
ENTITY_NAME  = N                    ENTITY_NAME  = N
ALIAS        = Serial number        ALIAS        = Cost
INSTANCE 1                          INSTANCE 1
   KEYWORD      = Y                    KEYWORD      = N
   ENTITY       = CARS                 ENTITY       = CARS
```

Figure 4.4: DATA ELEMENTS FROM CARS ENTITY

flagged as being a redundant data element. It is determined that the two data elements are completely different. This time the new data element is changed to "Serial_Num." The final data element "Price" is input without any problems.

The relationships are input into the data dictionary next. The relationships are entered by making each of them a separate entity. The fields are created from the keys of the two entities that the relationship connects. The process that is used to create these new entities varies depending on whether the relationship is 1-1, 1-N, or N-M. If the relationship is 1-1 the entity consists of the keys from the two connecting entities, and the key for the relationship entity is both of the keys from the two connecting entities. If the relationship is 1-N, the new

```
NAME        = CAR_BOUGHT
DESCRIPTION = Which car
              was purchased
TYPE        = Character
LENGTH      = 20
ENTITY_NAME = Y

NAME        = Name
DESCRIPTION = Name of a
              person
TYPE        = Character
LENGTH      = 20
ENTITY_NAME = N
ALIAS       = Moniker
ALIAS       = Last name
ALIAS       = First name
INSTANCE 1
   KEYWORD      = Y
   ENTITY       = CLIENT
INSTANCE 2
   KEYWORD      = Y
   ENTITY       = SALESMAN
INSTANCE 3
   KEYWORD      = N
   ENTITY       = CAR_BOUGHT

NAME        = Serial_Num
DESCRIPTION = Unique factory
              given number
TYPE        = Alphanumeric
LENGTH      = 15
ENTITY_NAME = N
ALIAS       = Serial number
INSTANCE 1
   KEYWORD      = Y
   ENTITY       = CARS
INSTANCE 2
   KEYWORD      = Y
   ENTITY       = CAR_BOUGHT
```

Figure 4.5: DATA ELEMENTS FROM CAR_BOUGHT ENTITY

entity consists of the keys from the connecting entities and the key is the key from the member entity. If the relationship is N-M, the entity consists of both keys, and the key is a concatenation of the other two keys.

One entity is created for each relationship in the example. The entities can be seen in Figure 4.5 for the CAR_BOUGHT entity, and Figure 4.6 for the BOUGHT_FROM entity. The fields consist of the keys from the two connecting entities.

The CAR_BOUGHT entity is easily created because both entities it connects, CLIENT and CARS, have different data elements

for their keywords. The fields for CAR_BOUGHT are "Name" and "Serial_Num" with the latter being the key for the entity.

The BOUGHT_FROM entity is created next. The two entities it connects, CLIENT and SALESMAN, have the same data element as their keywords. Two new data elements have to be created for the BOUGHT_FROM entity for the relational data base to use the relation properly. The two new data elements that are created are "C_Name" for the Name from CLIENT, and "S_Name" for the Name from SALESMAN. They both have "Name" for an alias. The input interactive redundancy check flags these, but the flag is overridden by a human and the data elements are added to the dictionary. Even though the data elements appear to be redundant, they

```
NAME        = BOUGHT_FROM          NAME        = C_Name
DESCRIPTION = Person car was       DESCRIPTION = Name of client
              bought from                        for relation
TYPE        = Character            TYPE        = Character
LENGTH      = 20                   LENGTH      = 20
ENTITY_NAME = Y                    ENTITY_NAME = N
                                   ALIAS       = Name
                                   INSTANCE 1
                                      KEYWORD     = Y
                                      ENTITY      = BOUGHT_FROM

NAME        = S_Name
DESCRIPTION = Name of Salesman
              for relation
TYPE        = Character
LENGTH      = 20
ENTITY_NAME = N
ALIAS       = Name
INSTANCE 1
   KEYWORD     = N
   ENTITY      = BOUGHT_FROM
```

Figure 4.6: DATA ELEMENTS FROM BOUGHT_FROM ENTITY

are quite different semantically.

The data dictionary now has the information that is needed to dynamically create a data base schema.

4.5 CREATING THE DATA BASE

The data dictionary queries the user on what to call the new data base. In this case the data base is to be named "Automobile." The DD is now ready to create the data base schema.

The data dictionary searches through the data dictionary for all of the data elements that have "Y" in their ENTITY_NAME fields and puts them into a buffer. The data elements that match are:

CLIENT
SALESMAN
CARS
CAR_BOUGHT
BOUGHT_FROM

The data dictionary takes each entity name and processes it separately. The first entity name the DD processes is CLIENT. The dictionary takes the CLIENT entity name and searches for all the fields that it contains. The DD goes to the Instance entity in the data dictionary and looks at the ENTITY field for any instance of CLIENT. When there is a match, the DD stores the name of the data element and then looks at the KEYWORD field in the same instance. It is recorded if the data element is a keyword or not. In this case the DD will find these matching fields for CLIENT:

Name
Address

Credit

The only field marked for a keyword was the "Name" field, so it will be the key for the CLIENT entity.

The data dictionary now dynamically creates the entity. The DD takes the "Name" Element entity and locates the type and length. In this case it is a character of length 20. This is stored in a buffer. The next field is now accessed. When the data dictionary has all of the information it needs for the CLIENT entity fields, it will create the entity. Creating the entity is done with the following INGRES command:

```
create CLIENT (
Name = c20,
Address = c40,
Credit = c10)
```

The data dictionary allows the following types; character, alphanumeric, integer, or floating. However, INGRES does not have a corresponding type for alphanumeric. It will be entered

CLIENT		SALESMAN		CARS	
Name*	Char 20	Name*	Char 20	Model	Char 20
Address	Char 40	ID_Number	Char 20	Serial_Num*	Char 15
Credit	Char 10	Address	Char 40	Price	Float 4
		Salary	Char 11		

CAR BOUGHT		BOUGHT FROM	
Name	Char 20	C_Name*	Char 20
Serial_Num*	Char 15	S_Name	Char 20

* = keys

Figure 4.7: SAMPLE DATA BASE

into INGRES as a character field.

Assuming that the data dictionary has gone through the same process for each of the data base entities, the finished data base is seen in Figure 4.7. This data base is fully dependent on the data dictionary for its data definitions. When the data dictionary is used in its normal capacity, the data base and the data dictionary will have the exact same data definitions. Therefore, the data dictionary is dynamic in nature.

CHAPTER FIVE

DATA DICTIONARY DESIGN AND IMPLEMENTATION

5.1 PROGRAM MODULES

The data dictionary system is a menu driven interactive program. A diagram showing how the modules relate to one another can be seen in Figure 5.1. The rest of this chapter will be a discussion of each of these modules.

5.1.1 MAIN MODULE

The main module is the foundation of the interactive aspects of the data dictionary. Its main purpose is a menu allowing the user to choose a module to access. But it also prepares the sys-

```
                          -------------
                         |             |
                         |    MAIN     |
                         |             |
                          -------------
                                |
                                |
   --------------------------------------------------------------
   |            |              |              |              |
   |            |              |              |              |
 ----------   ----------   ----------   -----------    ---------
|          | |          | |          | |           |  |         |
| ADD      | | MODIFY   | | DELETE   | | LIST      |  | CREATE  |
| ELEMENT  | | ELEMENT  | | ELEMENT  | | ELEMENTS  |  | DATA    |
 ----------   ----------   ----------   -----------   | BASE    |
   |            |                                      ---------
   --------------
         |
   -------------
  |             |
  | REDUNDANCY  |
  |   CHECK     |
   -------------
```

Figure 5.1: DATA DICTIONARY MODULES

tem to interact with INGRES. The system accesses the data dictionary data base with the command:

```
## ingres dictionary
```

The "ingres" accesses the INGRES DBMS, and "dictionary" accesses the dictionary data base within INGRES. The entities for dictionary are initialized in the following manner:

```
## range of e is element
## range of s is synonym
## range of i is instance
```

The '##' at the start of the line signifies that the commands are for INGRES and are not regular C language commands. The second set of commands allows the system to refer to the entities by the single letters instead of the entire name.

After the system has been initialized, the opening menu is output:

```
Which of the following would you like to do?

1) Add a data element.
2) Modify a data element.
3) Delete a data element.
4) List out the data elements.
5) Create a data base schema in INGRES.
6) Exit the data dictionary.
```

If the user enters anything other than a 1,2,3,4,5, or 6, the system outputs the error message "Not an option" and the opening menu is output again.

5.1.2 ADD ELEMENT MODULE

The Add Element module allows a user to interactively enter the metadata for a data element. A diagram of the module can be

seen in Figure 5.2. When a user accesses the module, the user is shown the format of a normal data element:

```
This is the format of the data element relation.

NAME        = Character 20
DESCRIPTION = Character 30
TYPE        = Character 12
LENGTH      = Integer 2
ENTITY_NAME = Character 1  (Y/N)
ALIAS       = Character 15 (One or more)
INSTANCE                   (One or more)
   KEYWORD  = Character 1  (Y/N)
   ENTITY   = Character 20
```

The user is then asked to enter the metadata for the data element. The following prompts will appear one at a time:

```
Data Element NAME =
Data Element DESCRIPTION =
Data Element TYPE =
Data Element LENGTH =
Data Element ENTITY_NAME =

Do you want to input any aliases?(Y/N)
Data Element ALIAS =
Do you want to input another?(Y/N)

Do you want to input any instances?(Y/N)
Data Element KEYWORD =
Data Element INSTANCE =
Another one?(Y/N)
```

The "get_input" module is called each time the system needs to read an input. Get_input reads the keystrokes from the keyboard and stores them in a global character array.

The user is allowed to input any number of ALIASES and INSTANCES for a particular data element. The user is queried whether to input an ALIAS. If the response is "Y", the user is allowed to input the ALIAS. The system then queries the user whether another ALIAS will be input. This process will be

```
                    -----------
                    |         |
                    | ADD     |
                    | ELEMENT |
                    -----------
                         |
  -----------------------------------------------------
  |            |                |               |
---------  -------------  -----------  -----------
|       |  |           |  |         |  |         |
| GET   |  | REDUNDANCY|  | PRINT   |  | REMOVE  |
| INPUT |  | CHECK     |  | ELEMENT |  -----------
---------  -------------  -----------
```

Figure 5.2: ADD ELEMENT MODULE

repeated until the user responds with "N." The same process is followed to input an INSTANCE.

The Redundancy Check module is called every time that a NAME or ALIAS is input into the system. If the Redundancy Check module returns that the data element is redundant, the Add Element module outputs the following error message:

```
ERROR---A REDUNDANT DATA ELEMENT HAS BEEN INPUT!
IT IS BEGIN REMOVED FROM THE DATA DICTIONARY!
```

and removes the data element from the data dictionary using the "remove" module.

If the data element is valid, the system adds the data element to the data dictionary with the following commands:

```
##      append to element (name = ename, description = edescript,
##                        type = etype, length = elength,
##                        entity_name = eentity_name)

##      append to synonym (name = ename, alias = ealias)

##      append to instance (name = ename, keyword = ekeyword,
##                         entity = eentity)
```

In the first command, "name" is the name of the field in the element entity, and "ename" is the C language variable that contains the input from the user. When the entire element has been inserted, the system calls the "print_element" which outputs the entire element. The user can then input another element or return to the main module.

5.1.3 MODIFY ELEMENT MODULE

The Modify Element module takes a data element specified by the user and allows the user to modify it. A diagram of the Modify Element module can be seen in Figure 5.3. The user is first queried for the Name of the data element to modify:

```
What is the name of the element that you want to modify?
(If you don't know any elements type 'q').
```

If the user responds with 'q', the system calls the module "short_list" which outputs a list of the data element Names. The query is then repeated. When the user has specified a data element Name, the system displays the current version of the data element using the "print_element" module. Print_element uses the same format as Figure 3.4. The following menu is then outputted:

```
Which one do you want to change?
1)  NAME
2)  DESCRIPTION
3)  TYPE
4)  LENGTH
5)  ENTITY_NAME
6)  ALIAS
7)  ADD OR DELETE AN ALIAS
8)  INSTANCE
9)  ADD OR DELETE AN INSTANCE
```

If the user responds with any of the numbers from 1-5, the system will modify the element entity. The system queries "What do you want to change it to?" If the user had chosen to modify the type field, the system would use the following INGRES command:

```
##      replace e (type = etype) where
##      e.name = ename
```

It should be noted that to change the NAME field, the NAME fields in the Synonym and Instance entities are also changed to correspond with the new data element Name so the INGRES command

```
                    --------------
                    |            |
                    |  MODIFY    |
                    |  ELEMENT   |
                    --------------
                          |
 --------------------------------------------------------------
 |            |             |                  |              |
---------  ---------  -------------  ----------------  -----------
|       |  |       |  |           |  |              |  |         |
| GET   |  | SHORT |  |  PRINT    |  |  REDUNDANCY  |  | REMOVE  |
| INPUT |  | LIST  |  |  ELEMENT  |  |  CHECK       |  -----------
---------  ---------  -------------  ----------------
```

Figure 5.3: MODIFY ELEMENT MODULE

is a three step process. This presented a problem in that the new name was read into the ename variable, so there was no way to match with the Element, Synonym, and Instance entities. A new variable "old_ename" was introduced to hold the old name so it could be used for the comparison. The following set of commands was then used:

```
##      replace e (name = ename) where
                     e.name = old_ename
##      replace s (name = ename) where
                     s.name = old_ename
##      replace i (name = ename) where
                     i.name = old_ename
```

If the user responds with 6 or 7, the Synonym entity will be modified. For 6, the system first queries for which Alias to change because there may be several ALIASES. The modified Alias name is then asked for.

For 7, the system first queries the user whether to add or delete an Alias. The system asks either for the new Alias or for the Alias to delete.

If the user inputs number 8, the system outputs the following query:

```
Which instance do you want to change?
(Signify by the name of the entity)

Which do you want to change?
1) KEYWORD
2) ENTITY

What do you want to change it to?
```

Signifying an instance to change by the name of the entity field is not the ideal way of specifying. It would be better to choose

by the Instance number, but this can not be done because the instances are not stored by INGRES in any particular order.

For 9, the system first finds out if the user wishes to add or delete an Instance. The system asks either for the new KEYWORD and ENTITY or which of the Instances to delete.

When the user modifies the NAME or ALIAS field, the Redundancy Check module is called to check the modified field for data redundancies. If either is found redundant, the user is queried whether they wish to keep the data element the way it was or use the modified version. The system outputs:

```
ERROR---A REDUNDANT DATA DATA ELEMENT HAS BEEN INPUT!
IT IS BEING REMOVED FROM THE DATA DICTIONARY!

Do you want to keep this element the way it was?(Y/N)
(No will delete the element)
```

If the user wants to use the modified version, the data element is immediately removed from the data dictionary via the remove module.

5.1.4 REDUNDANCY CHECK MODULE

The Redundancy Check module takes the data element field sent to it (either NAME or ALIAS), and compares it to all of the resident Names and Aliases in the data dictionary. If the new one matches anywhere in the system, the calling process is told that there was a match and an error message is output. Otherwise, the calling process is told that there were no matches.

The comparisons are made with the following INGRES commands:

```
##      retrieve (fill = e.name) where e.name = word
##      {
```

```
                found = true;
##      }
##      retrieve (fill = s.alias) where s.alias = word
##      {
                found = true;
##      }
```

"Fill" is a filler to temporarily store the matched data element Name or Alias. "Word" is either the Name or Alias that was sent to the module to be checked for redundancy. "Found" is preset to false, and sent back to the calling module with either true or false.

The system searches through the entities and goes into the compound statement only if the word matches with either of the data element Names or data element Aliases.

5.1.5 DELETE ELEMENT MODULE

The Delete Element module will delete the data element that the user specifies. A diagram of the Delete element module can be seen in Figure 5.4. The system outputs are:

Do you want to see a list of the elements?(Y/N)

What is the name of the element that you want to delete?

Are you sure that you want to delete the <name> element?(Y/N)

The module first queries the user to see if a list of data element names is desired. If so, the "short_list" module is called. The system then queries for the Name of the data element to delete. If a valid data element Name is input, the system queries the user if he is sure that he wants to delete the data element that has been input. If the response is "no," then the

```
                 -------------
                 |           |
                 |  DELETE   |
                 |  ELEMENT  |
                 -------------
                       |
        ----------------------------
        |              |              |
   ----------    -----------    -----------
   |        |    |         |    |         |
   |  GET   |    |  SHORT  |    |  REMOVE |
   |  INPUT |    |  LIST   |    -----------
   ----------    -----------
```

Figure 5.4: DELETE ELEMENT MODULE

user is sent back to the main module. Otherwise, the data element is deleted from the data dictionary by the "remove" module. Remove does this in a three part process. First the system Element entity is accessed and matched on the Name field, and the matching tuple is deleted. Second, the system accesses the Synonym entity and matches on the Name field and deletes the matching tuples. There may be several Synonym tuples that need to be deleted. Third, the system accesses the Instance entity and matches on the Name field and deletes the matching tuples. The INGRES commands look like:

```
##      delete e where e.name = word
##      delete s where s.name = word
##      delete i where i.name = word
```

The "word" is the name of the data element to be deleted that was sent from the Delete Element module.

5.1.6 LIST ELEMENTS MODULE

The List Elements module takes each data element in the data dictionary, and individually outputs it in the format shown in Figure 3.4. A diagram of the module can be seen in Figure 5.5. The system displays one data element on the screen using the "print_element" module. Then the system waits until the user hits the return key to display the next one. When the data elements in the data dictionary have been shown, the user is sent back to the main module.

5.1.7 CREATE DATA BASE MODULE

Before the system can create the data base schema, the user needs to create the data base that he wishes to use. This is done with the following command:

creatdb <data base name>

This command is done in UNIX, not in INGRES.

The Create data base module first queries the user whether a data base has been created. If the user responds "n," then the user is sent back to the main module. Otherwise the system

```
 --------------
|              |
|   LIST       |
|   ELEMENT    |
 --------------
       |
 --------------
|              |
|   PRINT      |
|   ELEMENT    |
 --------------
```

Figure 5.5: LIST ELEMENT MODULE

queries the user for the name of the data base. The output looks like:

Did you create a data base?(Y/N)

What is the name of the data base?

When the system has received a valid name for the data base, it goes through the process that was outlined in Chapter Four.

The system first accesses the Element entity in the data dictionary. It retrieves the tuples where the ENTITY_NAME field is 'Y.' It takes each one of these tuples individually and stores the NAME field in an array.

Next the system accesses the Instance entity. The entity names are taken one at a time from their arrays and used to retrieve tuples from the Instance entity where they match the entity name. The NAME and KEYWORD for the retrieved tuple are stored in the array.

The system next accesses the Element entity again. The field names are separately accessed in their arrays and used to retrieve the TYPE and LENGTH where the NAME matches with the field name. These are stored in the same array. The data dictionary now has the information needed to create the data base.

The data dictionary will access one of the entity names from its array, and withdraw its corresponding elements from their array. The information will be put into an INGRES command in the following format:

```
create <entity name> (
<field> = <type(either c, i, or f)><length>)
```

When the entities have been created, the system will output:

The data base schema for <name> has been created.

The user then exits from the data dictionary.

5.2 SAMPLE DATA BASE CREATION

To illustrate how the system creates a data base, a simple example will be discussed. The ER diagram for this example can be seen in Figure 5.6.

The data dictionary for this example has already been input into the system and can be seen in Figure 5.7. It is assumed that there were no problems with any data redundancies. For simplicity, aliases have been omitted.

The first thing the system does is to search the data dictionary data base for data elements that are entity names. The names of these data elements are stored in an array called entities. The commands that do this are:

```
##      retrieve (ename = e.name) where e.entity_name = 'y'
##      {
                entities[x] = ename;
                x++;
##      }
```

In these commands, the "e.name" is the name field in the element entity, and the "e.entity_name" is the entity_name field in the

```
---------------                    -------------------
| CLIENT       |1     HAS        N| ACCOUNT           |
|   Name       |------------------|   Account_type    |
---------------      ACCOUNT       -------------------
```

Figure 5.6: EXAMPLE TWO ER DIAGRAM

element entity. The system will automatically repeat this section until it has found the entity_names that match. The entities array now contains the data element names that are entity names. They are:

```
CLIENT
ACCOUNT
HAS_ACCOUNT
```

The system next goes to the instance entity and retrieves

```
NAME        = CLIENT                NAME        = ACCOUNT
DESCRIPTION = Buyer of              DESCRIPTION = Thing purchased
              services                            by Client
TYPE        = Character             TYPE        = Character
LENGTH      = 20                    LENGTH      = 15
ENTITY_NAME = Y                     ENTITY_NAME = Y


NAME        = Name                  NAME        = Account_No
DESCRIPTION = Name of a             DESCRIPTION = Unique ID of
              person                              Account
TYPE        = Character             TYPE        = Alphanumeric
LENGTH      = 20                    LENGTH      = 10
ENTITY_NAME = N                     ENTITY_NAME = N
INSTANCE 1                          INSTANCE 1
   KEYWORD    = Y                      KEYWORD    = Y
   ENTITY     = CLIENT                 ENTITY     = ACCOUNT
INSTANCE 2                          INSTANCE 2
   KEYWORD    = Y                      KEYWORD    = Y
   ENTITY     = HAS_ACCOUNT            ENTITY     = HAS_ACCOUNT


NAME        = HAS_ACCOUNT
DESCRIPTION = Accounts client
              has
TYPE        = Character
LENGTH      = 20
ENTITY_NAME = Y
```

Figure 5.7: EXAMPLE TWO DATA DICTIONARY

the data element names that are instances of one of the entity names. It stores the entity name, matching data element name, and keyword in an array called entity_buffer. These are stored in the following manner:

```
##      retrieve (ename = i.name, ekeyword = i.keyword)
##               where i.entity = entities[x]
##      {
                 entity_buffer[y][0] = entities[x];
                 entity_buffer[y][1] = ename;
                 entity_buffer[y][3] = ekeyword;
##      }
```

After this step is completed, the entity_buffer contains the following information:

```
entity_buffer[0] = CLIENT, Name, Y
             [1] = ACCOUNT, Account_No, Y
             [2] = HAS_ACCOUNT, Name, Y
             [3] = HAS_ACCOUNT, Account_No, Y
```

The system now knows that there will be three different entities in the new data base schema: CLIENT, ACCOUNT, and HAS_ACCOUNT.

The system next accesses the Element entity again to get the types and lengths of the data elements. These are concatenated and stored in the entity_buffer array. The command looks like the following example:

```
##      retrieve (etype = e.type, elength = e.length)
##               where ename = entity_buffer[x][y]
##      {
                 if ((etype == "Character")||
                    (etype == "Alphanumeric"))
                     entity_buffer[x][y][0] = 'c';
                 if (etype == "Integer")
                     entity_buffer[x][y][0] = 'i';
                 if (etype == "Floating")
                     entity_buffer[x][y][0] = 'f';
                 entity_buffer[x][y][1] = elength;
##      }
```

This set of commands will put either c, i, or f in the first place in the array, and the length in the rest of the slot. After this command, the entity_buffer will appear as follows:

```
entity_buffer[0] = CLIENT, Name, c20, Y
             [1] = ACCOUNT, Account_No, c10, Y
             [2] = HAS_ACCOUNT, Name, c20, Y
             [3] = HAS_ACCOUNT, Account_No, c10, Y
```

The system now has the information that it needs to create the data base schema. The schema will be created with the following three commands:

```
create entity_buffer[0][0](
entity_buffer[0][1] = entity_buffer[0][2])

create entity_buffer[1][0](
entity_buffer[1][1] = entity_buffer[1][2])

create entity_buffer[2][0](
entity_buffer[2][1] = entity_buffer[2][2],
entity_buffer[3][1] = entity_buffer[3][3])
```

The data base schema will be:

```
CLIENT             ACCOUNT
Name char 20       Account_No char 10

HAS ACCOUNT
Name       char 20
Account_No char 10
```

CHAPTER SIX

SUMMATION AND FUTURE WORK

6.1 SUMMATION

Data dictionaries are a data base software tool that are gaining more acceptance. Their functions include:

- Storage of metadata definitions
- Minimization of data redundancy in the data base
- Data and relationship handling
- Added security to the data base

There are two basic types of data dictionaries, static and dynamic. A static DD does not participate actively in the handling of data base transactions. A dynamic data dictionary does. Dictionaries can also be either DBMS-dependent, requiring a particular DBMS to operate with, or stand-alone, operating without the aid of any particular DBMS.

Data dictionaries have been used primarily as a data base documentational tool. Recently it has been suggested that the DD be used in the design process of the data base. Used in the design process, the data definitions can be input into the DD to insure that they meet non-redundancy requirements before the data base is finalized. Doing so ensures that the data base is fully dependent on the data dictionary for its data definitions. Other programs and reports could be added to the system but they would have to use the metadata already resident in the data dictionary. This type of environment would ensure that the data dictionary is dynamic at data base and application program design time.

This concept has been taken a step further by this work. The data definitions are entered into the data dictionary during the

data base design phase as previously stated, but then the data dictionary dynamically creates a data base schema from the metadata. Thus, the metadata that is input into the data dictionary is the exact same metadata that is in the final data bese schema.

This data dictionary has been designed with little outside information on the implementation level because little information is available to aid in the implementation of a data dictionary with dynamic characteristics.

The data dictionary has been designed using the following components:

1) Development of a data dictionary data base to store the metadata.

2) Development of interactive input programs.

3) Development of an interactive output program.

4) Development of the techniques used to design and implement the dynamic creation of a data bese schema.

6.2 FUTURE WORK

There are many improvements that could be made to this data dictionary. However, the DD is designed as the base of a system that could become a fully dynamic data dictionary.

Currently no semantic checks are done during the data redundancy checking process. Semantic checks could be done by checking the DESCRIPTION of the data elements. Additional use of artificial intelligence techniques such as an expert system or an extensive knowledge based system using conceptual graphs could be added to allow more effective checking. The actual checking

could involve:

1) Requiring an organizational wide stylistic format for the data descriptions.

2) Using a natural language query system to compare descriptions.

3) Using some form of 1 and 2 together.

A second improvement would be to add a dynamic data directory with the data dictionary. A directory would give the user the capability to detect at execution time where each of the data elements is being used.

A third improvement would be to require users to go through the data dictionary to access INGRES. This data dictionary system will allow a user to bypass the DD and go directly into INGRES making it difficult for an organization to enforce the data definitions throughout the system.

A fourth improvement would be to integrate other parts of the system into the data dictionary. These may include the operating system and language compilers.

WORKING BIBLIOGRAPHY

[A Survey 77]
A Survey of Eleven Government-Developed Data Element Dictionary/Directory Systems NBS Special Publication 500-16 (1977).

[Allen 82]
Allen, F.W., Loomis, M.E., and Manning, M.V., "The Integrated Dictionary/Directory System", ACM Computing Surveys, Vol. 14 (June 1982), 245-275.

[Asscher 84]
Asscher, Yvette. "Describing Businesses With Data Dictionaries," Data Processing, Vol. 26, No. 6 (July/August 1984), 17-19.

[Baker 76]
Baker, G.J. "Database and Teleprocessing Software Review," Database Journal, Vol. 6, No. 12 (1976), 2-7.

[Berg 81]
Berg, John L., Graham, Marc, and Whitney, Kevin. Database Architectures-- A Feasibility Workshop Report. NBS Special Publication 500-76, 1981.

[Cahill 70]
Cahill, John J. "A Dictionary/Directory Method for Building a Common MIS Data Base," Journal of Systems Management, Vol. 2, No. 1 (November 1970), 23-29.

[Collard 74]
Collard, Albert F. "A Data Dictionary Directory," Journal of Systems Management, Vol. 25, No. 6 (June 1974), 22-25.

[Curtice 81]
Curtice, Robert M. "DATA DICTIONARIES: An Assessment of Current Practice and Problems," Proceedings Seventh International Conference on Very Large Data Bases, Cannes France (1981).

[Curtice 74]
Curtice, Robert M. "Some Tools for Database Development," Datamation, Vol 20. (July 1974), 102-104.

[Davis 84]
Davis, Richard K. "New Tools and Techniques to make Data Base Professionals More Productive," Journal of Systems Management, Vol. 35, No. 6 (June 1984), 20-25.

[Deutch 79]
Deutch, Donald R. Modeling and Measurement Techniques for Evaluation of Design Alternatives in the Implementation of Database Management Software. NBS Special Publication 500-49 (1979).

[Durrell 84]
Durrell, William. "Disorder to Discipline Via the Data Dictionary," Journal of Systems Management, Vol. 34 (May 1984), 12-19.

[Epstein 77]
Epstein, Robert. A Tutorial on INGRES. Memorandum No. ERL-M44-25, Electronics Research Laboratory, College of Engineering, University of California, Berkeley (1977).

[Hardy 77]
Hardy, T., Leong-Hong, B., and Fife, Dennis. Software Tools: A Building Block Approach. NBS Special Publication 500-14 (1977).

[Hotada 77]
Hotada, R., and Tsubake, M.. "Self-Descriptive Relational Data Base," Proceedings Third International Conference of Very Large Data Bases, New York (1977).

[INGRES 85]
Introduction To INGRES. Alameda, California: Relational Technology Inc. (1985).

[Jardine 76]
Jardine, D.A. editor. The ANSI/SPARC DBMS Model. New York: North-Holland Publishing Company (1976).

[Jardine 84]
Jardine, D.A. "Concepts and Terminology for the Conceptual Schema and the Information Base," Computers and Standards, Vol. 3, No. 1 (1984) 3-17.

[King 81]
King, Judy. Evaluating Data Base Management Systems. New York: Van Nostrand Reinhold Company (1981).

[Konsynski 84]
Konsynski, Benn R. "Advances in Information System Design," Jounal of Management Information Systems, Vol. 1, No. 3 (Winter 1984-85), 5-32.

[Leong-Hong 82]
Leong-Hong, B., and Plagman, B. Data Dictionary/Directory Systems: Administration, Implementation, and Usage. New York: John Wiley and Sons (1982).

[Leong-Hong 77]
Leong-Hong, B., and Marron, B. "Technical Profile of Seven Data Element Dictionary/Directory Systems," NBS Special Publication 500-3 (1977).

[Marti 84]
Marti, Robert W. "Integrating Database and Program Descriptions Using an ER-Data Dictionary," The Journal of Systems and Software, Vol. 4, No. 2/3 (July 1984), 185-195.

[Martin 83]
Martin, James. Managing the Data Base Environment. Englewood Cliffs, New Jersey: Prentice-Hall (1983).

[Phillips 83]
Phillips, Robert W. Dynamic Data Dictionary. Master's Thesis at Kansas State University (1983).

[Recommend 79]
Recommendations for Database Management Standards. NBS Special Publication 500-51 (1979).

[Ross 81]
Ross, Ronald G. Data Dictionaries and Data Administration. New York: AMACOM (1981).

[Ruspini 84]
Ruspini, Enrique H., and Fraley, Robert. "ID: An Intelligent Information Dictionary System," The Journal of Systems Software, Vol. 4, No. 2/3 (July 1984), 197-205.

[Uhrowczik 73]
Uhrowczik, P.P. "Data Dictionary/Directories," IBM Systems Journal, Vol. 12, No. 5 (1973), 332-350.

[Van Duyn 84]
Van Duyn, Julie. "Data dictionaries as a Tool to Greater Productivity," Data Processing, Vol. 26, No. 6 (July/August 1984), 14-16.

[Van Duyn 82]
Van Duyn, Julie. Developing A Data Dictionary System. Englewood Cliffs, New Jersey: Prentice-Hall (1982).

[Wearing 73]
Wearing, Michael L. "Upgrade Documentation with a Data Dictionary," Computer Decisions, Vol. 5, No. 8 (August 1973), 29-31.

[Woodfill 85]
Woodfill, John, et. al. INGRES Reference Manual. Version 7 (1985).

[Zahran 81]
Zahran, F.S. "A Basic Structure for Data Dictionary Systems," ACM European Regional Conference (England) Proceedings... Systems Architecture. (March, 1981).

MISSING PAGE

APPENDIX ONE

MANUAL FOR DATA DICTIONARY

by Loren Wilson

This is a manual for the data dictionary with the INGRES DBMS. This data dictionary will allow data definitions to be added, modified, or deleted from the data dictionary. Once all of the data definitions have been entered, the data dictionary can create a relational data base schema in INGRES from the resident data definitions. The data dictionary program is a menu driven program for ease of use.

CREATING THE DATA DICTIONARY

Whenever a new data dictionary is wanted, one creates a copy of the dictionary data base using the following UNIX command:

copydb dictionary <data dictionary name>

This command will allow the data dictionary program to store the data definitions in the <data dictionary name>.

STARTING THE DATA DICTIONARY

After the data dictionary data base has been created, one is now ready to access the data dictionary program. The program to run the data dictionary is stored in a file called "dictionary.q". The program is

compiled to run in the following manner:

```
equel dictionary.q
cc dictionary.c -lq
a.out
```

When the input a.out is created, the data dictionary program will be started. The first thing that the data dictionary will ask for is the name of the data dictionary data base created earlier. This will be the same as the <data dictionary name> that was copied. When one gives it the name of a data dictionary data base, the data dictionary will access that data base, and store all of the data definitions there.

When the data dictionary has accessed a particular data base, it will then show you the opening menu:

Which of the following would you like to do?

1) Add a data element.
2) Modify a data element.
3) Delete a data element.
4) List out the data elements.
5) Create a data base schema in INGRES.
6) Exit the data dictionary.

ADDING A DATA ELEMENT

Adding a data element is done with option number 1 from the main menu.

Data element definitions are added to the data dictionary interactively. The data dictionary queries one with the name of the field, and waits for the input. Theses are the fields that are available for

each data element.

NAME-

the unique name of the data element. This must be of type character and less then 20 characters in length. NOTE: if a data element name is entered that already exists in the data dictionary as either a data element name or a data element alias, then the data dictionary will not allow the new data element name to added to the data dictionary data base. This happens when one gets the following error message:

```
ERROR---A REDUNDANT DATA ELEMENT HAS BEEN INPUT!
IT IS BEING REMOVED FROM THE DATA DICTIONARY!
```

DESCRIPTION-

a detailed english definition of the data element. It can be up to 30 characters in length.

TYPE-

the type of the data element. This <u>MUST</u> be one of the following: Character, Integer, Floating, or Alphanumeric. Nothing else will be accepted for this field.

LENGTH-

the maximum length of the data element. This can include the maximum number of characters allowed for the data element, or the maximum size of an

integer. This field must be an integer.

ENTITY_NAME-

this is a one character field with either a "Y" or "N", signifying that this data element is the name of an entity in the data base.

You will be asked if you wish to enter an ALIAS. If you answer "y", the data dictionary will expect you to input the following.

ALIAS-

another name that the data element may be known as. There may anywhere from zero to hundreds of these listed for a particular data element. NOTE: it is possible to enter a redundant data element ALIAS. When this happens, you will have the option of adding the data element to the data dictionary data base. The following error message is output:

```
WARNING!!! the alias input is redundant. ARE YOU SURE
THAT YOU WANT TO INPUT IT?(y/n)
(yes will delete the entire data element)
```

You will be asked if you wish to enter an Instance (a particular instance in the data base of this data element). If you answer "y", the data dictionary will expect you to input the following two fields. NOTE: if the ENTITY_NAME field for a data element is marked "Y", then that data element should not have any

Instances.

KEYWORD-

a one character field with either a "Y" or "N", signifying whether this instances of the data element is the key for an entity.

ENTITY-

the name of the entity that this instance of the data element belongs to. This is a field of 20 characters.

MODIFING A DATA ELEMENT

Modifing a data element is done from option number 2 from the main menu.

Any data element in the data dictionary can be modified. The first thing the dictionary will do is to query you for the name of the data element to be modified. If one is unsure of what the names of the data elements are, input a '?', and the dictionary will output a list of the data element names. When a data element name has been input, the dictionary will output the following menu:

```
Which one do you want to change.
1)  NAME
2)  DESCRIPTION
3)  TYPE
4)  LENGTH
5)  ENTITY_NAME
6)  ALIAS
7)  ADD OR DELETE AN ALIAS
8)  INSTANCE
9)  ADD OR DELETE AN INSTANCE
```

After input of the number of the thing that is to be changed, the dictionary will ask for the change. NOTE: whenever the NAME or ALIAS is changed the same redundancy checks are employed, and if the modified NAME or ALIAS is redundant, the same error messages will be output.

If one wishes to modify the ALIAS or INSTANCE, the dictionary will query for the particular one to modify, because the data dictionary data base can contain several of these for a single data element.

It is possible to add or delete ALIASES or INSTANCES from the data dictionary data base. These are done with options 7 and 9.

DELETING A DATA ELEMENT

Deleting a data element is done with option number 3 from the main menu.

Deleting a data element from the data dictionary data base is a simple task. You give the dictionary the name of the data element that you wish to delete, and it removes the entire data element.

LISTING OUT THE DATA ELEMENTS

Listing of the data elements is done with option number 4 from the main menu. If one wants to see what data elements are in the data dictionary data base and what they look like, this option will allow that.

CREATING A DATA BASE SCHEMA

A relational data base schema will be created from the data definitions resident in the data dictionary data base with option number 5 in the main menu.

When you entered all of the data element definitions into the data dictionary data bese, you can have the data dictionary automatically create a relational schema for you in a particular data bese. Before schema is created, one needs to have the use of a data base within INGRES for the schema to reside in. A data base can be created with the following command:

creatdh <data base name>

The data dictionary will ask a data base has been created, and what the name of it is. Then, it will automatically create a relational schema in it from the data definitions that have been input into the data dictionary.

NOTE: a proper schema will not be created from random data definitions. A data base should be properly designed before any of the data definitions are input into the data dictionary data base.

EXITING THE DATA DICTIONARY

The data dictionary can be exited by specifying the number 6 option in the main menu.

MISSING PAGE

APPENDIX TWO

```
/************************************************************/
/*                                                          */
/*  This program is intended to make the data dictionary    */
/*  in the INGRES DBMS interactive.  This program will      */
/*  allow anyone to interactively manipulate the data       */
/*  dictionary data base without having a working know-     */
/*  ledge of QUEL.  To run this program, it must first be   */
/*  run through the EQUEL proprocessor.  This can be done   */
/*  in the following manner:                                */
/*                  equel dictionary.q                      */
/*                  cc dictionary.c -lq                     */
/*                  a.out                                   */
/*                                                          */
/************************************************************/

#include <stdio.h>
#define maxsize 31
#define max_elements 100

char input[maxsize];

/* The following variables can be used directly with INGRES*/

##      char dict_name[21];
##      char ename[21];
##      char old_ename[21];
##      char edescript[31];
##      char etype[13];
##      char elength[4];
##      char eentity_name[2];
##      char ealias[16];
##      char old_alias[16];
##      char ekeyword[2];
##      char eentity[21];
##      char old_eentity[21];
##      char word[21];
##      char fill[21];

main()
{
        int x, done;
        extern char input[];

        printf("This program is the data dictionary to run0);
        printf(" with the INGRES DBMS.  It is designed to 0);
        printf("Add, Delete, or Modify data elements from 0);
        printf("a data dictionary.  The dictionary can be 0);
        printf("one that is already in existence, or one 0);
        printf("that you want to create.  0);
        printf("What is the name of the dictionary that 0);
        printf("you are using?0);
        get_input();
```

```
        x = 0;
        while ((dict_name[x] = input[x])!= ' ')
                x++;
        for (x = 1; x < 5; x++)
                printf("0);

##      ingres dict_name

##      range of e is element
##      range of s is synonym
##      range of i is instance

        while (done == 0)
        {
                printf("0);
                printf("Which of the following would you ");
                printf("like to do?0);
                printf("1) Add a data element.0);
                printf("2) Modify a data element.0);
                printf("3) Delete a data element.0);
                printf("4) List out the data elements.0);
                printf("5) Create a base schema in INGRES.0);
                printf("6) Exit the data dictionary.0);
                printf("0);
                get_input();
                if (input[1] != ' ')
                {
                        printf("Not an option0);
                }
                else
                {
                        switch(input[0])
                        {
                                case '1':
                                        add_element();
                                        break;
                                case '2':
                                        modify_element();
                                        break;
                                case '3':
                                        delete_element();
                                        break;
                                case '4':
                                        list_elements();
                                        break;
                                case '5':
                                        create_db();
                                        done = 1;
                                        break;
                                case '6':
                                        done = 1;
                                        break;
                                default:
                                    printf("Not an option0);
```

```
                                        break;
                                }
                        }
                }
##              exit
}

/*************************************************************/
/**********                                         ********/
/**********        GET_INPUT   and   YES_NO          ********/
/**********                                         ********/
/*************************************************************/

/* GET_INPUT reads in whatever the user has input, and  */
/* places it in a global variable called input. */

get_input()
{
        extern char input[];
        int x;
        char c;

        for (x = 0; x <= maxsize; x++)
                input[x] = ' ';
        for (x = 0; (c = getchar()) != '0; x++)
                input[x] = c;
        input[x] = ' ';
}

/* The function yes_no looks for either a 'y' or 'n' */
/* showing if the user means yes or no              */
yes_no()
{
        int done, loop;

        loop = 0;
        while (loop == 0)
        {
                get_input();
                if (input[0] == 'n' || input[0] == 'N')
                {
                        loop = 1;
                        return(1);
                }
                else
                {
                        if(input[0] == 'y'||input[0]=='Y')
                           return(0);
                        else
                        {
                           printf("'y' for yes, ");
                           printf("'n' for no.0);
                        }
                }
```

```
        }
}

/************************************************************/
/****************                            *************/
/****************        LIST_ELEMENTS       *************/
/****************                            *************/
/************************************************************/

list_elements()
{
        char buffer[max_elements][21];
        int y, x;

        y = 0;
        printf("0);
        printf("0);
        printf("Here are the present elements in the ");
        printf("dictionary.0);
##      retrieve (ename = e.name)
##      {
                y++;
                x = 0;
                while((buffer[y][x] = ename[x])!= ' ')
                        x++;
##      }
        for (x = 1; x <= y; x++)
        {
                print_element(buffer[x]);
                printf("To continue hit <return>0);
                printf("0);
                get_input();
        }
}

/************************************************************/
/*************                               **************/
/*************        PRINT_ELEMENT          **************/
/*************                               **************/
/************************************************************/

print_element(ename)

{
        int count, x, there;

        count = 0;
        there = 0;
        printf("0);
        printf("This is what the %s ", ename);
        printf("element looks like now.0);

##      retrieve (edescript = e.description, etype = e.type,
##      elength = e.length, eentity_name = e.entity_name)
```

```
##      where e.name = ename
##      {
                there = 1;
                printf("NAME =              %s0, ename);
                printf("DESCRIPTION =       %s0, edescript);
                printf("TYPE =              %s0, etype);
                printf("LENGTH =            %s0, elength);
                printf("ENTITY_NAME =       %s0, eentity_name);
##      }

##      retrieve (ealias = s.alias) where s.name = ename
##      {
                printf("ALIAS =             %s0, ealias);
                count++;
##      }

        x = 0;
##      retrieve (ekeyword = i.keyword, eentity = i.entity)
##      where i.name = ename
##      {
                x++;
                printf("INSTANCE %d0, x);
                printf("   KEYWORD =        %s0, ekeyword);
                printf("   ENTITY =         %s0, eentity);
##      }
        if (there == 0)
        {
                printf("The element is not in the ");]
                printf("data dictionary.0);
        }
        return(count);
}

/************************************************************/
/********                                   *************/
/********           ADD_ELEMENT             *************/
/********                                   *************/
/************************************************************/

add_element()

/* The purpose of this function is to add data elements */
/* to the data dictionary.                              */
{
        extern char input[];
        int go, x, number, loop, repeat, done, accept;
        int again, finished;

        finished = 0;
        for (x = 1; x < 10; x++)
                printf("0);
        printf("This is the format of the data element ");
        printf("relation0);
        printf("NAME        = Character 200);
```

```
printf("DESCRIPTION = Character 300);
printf("TYPE        = Character 120);
printf("LENGTH      = Integer 20);
printf("ENTITY_NAME = Character 1 (Y/N)0);
printf("ALIAS       = Character 15 (One or more)0);
printf("INSTANCE                   (One or more)0);
printf("   KEYWORD  = Character 1  (Y/N)0);
printf("   ENTITY   = Character 200);
printf("0);
repeat = 0;
while (repeat == 0)
{

        printf("Data Element NAME =     ");
        get_input();
        x = 0;
        while ((ename[x] = input[x]) != ' ')
              ++x;
        accept = redundancy_check(ename);
        if (accept != 1)
        {
           printf("0);

           printf("Data Element DESCRIPTION =   ");
           get_input();
           x = 0;
           while ((edescript[x] = input[x]) != ' ')
                 ++x;
           printf("0);

           again = 0;
           while (again == 0)
           {
                printf("Data Element TYPE =     ");
                get_input();
                /* Make sure that the type is in  */
                /* the proper format so the create*/
                /* db function can work properly  */
                if((compare_strings(input,"Character"
                ,9)==1)&&(compare_strings(input,
                "Alphanumeric",12)==1)&&
                (compare_strings(input,"Integer",7)
                ==1)&&(compare_strings(input,
                "Floating",8)==1))
                {
                   printf("0he type must be ");
                   printf("either:0);
                   printf("     Character0);
                   printf("     Alphanumeric0);
                   printf("     Integer0);
                   printf("     Floating0);
                   printf("0ther input will not ");
                   printf("be accepted.0);
                   printf("0);
```

```
                  }
                  else
                     again = 1;
               }
               x = 0;
               while ((etype[x] = input[x]) != ' ')
                    ++x;
               printf("0);

               printf("Data Element LENGTH =    ");
               get_input();
               x = 0;
               while((elength[x] = input[x]) != ' ')
                    ++x;
               printf("0);

               printf("Data Element is ENTITY_NAME ");
               printf("(Y/N) =    ");
               get_input();
               x = 0;
               while((eentity_name[x]=input[x])!=' ')
                    ++x;

##             append to element (name = ename,
##             description = edescript, type = etype,
##             length = elength, entity_name =
##             eentity_name)

               printf("0);
               printf("Do you want to input any");
               printf(" alias's?(y/n)0);
               loop = yes_no();
               while (loop == 0)
               {
                    printf("Data Element  ALIAS =  ");
                    get_input();
                    x = 0;
                    while ((ealias[x] = input[x]) != ' ')
                           ++x;
                    accept = redundancy_check(ealias);
                    if (accept != 1)
                    {
##                     append to synonym (name = ename,
##                     alias = ealias)
                       printf("Another alias?(y/n)0);
                       loop = yes_no();
                    }
                    else
                    {
                       printf("WARNING!!! The alias ");
                       printf("input is redundant. /n");
                       printf("ARE YOU SURE THAT YOU ");
                       printf("WANT TO INPUT IT?");
                       printf("(y/n)0);
```

```
                    printf("(yes will delete the ");
                    printf("entire data element)0);
                    go = yes_no();
                    loop = 1;
                    if (go == 0)
                    {
                         remove(ename);
                         finished = 1;
                    }
                }
            }
            if (finished != 1)
            {
                printf("0);
                printf("Do you want to input any");
                printf(" instances?(y/n)0);
                done = yes_no();
                if (done == 0)
                {
                    loop = 0;
                    while (loop == 0)
                    {
                         printf("Data Element ");
                         printf("KEYWORD =    ");
                         get_input();
                         x = 0;
                         while((ekeyword[x] =
                         input[x]) != ' ')
                                 x++;
                         printf("0);

                         printf("Data Element ");
                         printf("INSTANCE =   ");
                         get_input();
                         x = 0;
                         while ((eentity[x] =
                         input[x]) != ' ')
                                 x++;
                         printf("0);

##                       append to instance (name =
##                       ename, keyword= ekeyword,
##                       entity = eentity)

                         printf("Another instance?");
                         printf(" (y/n)0);
                         loop = yes_no();
                    }
                }

                print_element(ename);

                printf("0);
                printf("0);
```

```
                        printf("Add another element?(y/n)0);
                        printf("0);
                        repeat = yes_no();
                        printf("0);
                    }
                    else
                        repeat = 1;
                }
                else
                    repeat = 1;
        }
}

/***********************************************************/
/************                        ********************/
/************       MODIFY_ELEMENT    ********************/
/************                        ********************/
/***********************************************************/

modify_element()

/* This function is to modify any elements */
/* in the data dictionary */

{
        extern char input[];
        int finished, x, loop, count, what;
        int accept, why, again;
        char number;

        for (x = 1; x < 6; x++)
                printf("0);
        loop = 0;
        while (loop == 0)
        {
           loop = 1;
           printf("What is the name of the element that ");
           printf("you want to modify?0);
           printf("(If you don't know any elements type ");
           printf("'?').0);
           get_input();
           if (input[0] == '?')
           {
                   loop = 0;
                   short_list();
           }
           printf("0);
        }
        printf("0);
        printf("0);
        x = 0;
        while ((ename[x] = input[x]) != ' ')
                x++;
        count = print_element(ename);
```

```
printf("To continue hit <return>0);
get_input();
loop = 0;
while (loop == 0)
{
        loop = 1;
        printf("0);
        printf("0);
        printf("Which one do you want to change.0);
        printf("1)  NAME0);
        printf("2)  DESCRIPTION0);
        printf("3)  TYPE0);
        printf("4)  LENGTH0);
        printf("5)  ENTITY_NAME0);
        printf("6)  ALIAS0);
        printf("7)  ADD OR DELETE AN ALIAS0);
        printf("8)  INSTANCE0);
        printf("9)  ADD OR DELETE AN INSTANCE0);
        get_input();
        number = input[0];
        if (number == '6')
        {
           printf("Which alias do you want to ");
           printf("change?0);
           get_input();
           printf("0);
           x = 0;
           while ((old_alias[x] = input[x]) != ' ')
                x++;
        }
        if (number == '8')
        {
           printf("0);
           printf("Which instance do you want ");
           printf("to change?0);
           printf("(Signify by the name of the ");
           printf("entity)0);
           get_input();
           x=0;
           while((old_eentity[x]=input[x])!=' ')
                x++;
        }
        if((number!='7')&&(number!='9')&&
        (number!='8'))
        {
           printf("What do you want to change ");
           printf("it to?0);
           get_input();
        }
        printf("0);
        switch (number)
        {
           case '1':
                finished = 0;
```

```
                        x = 0;
                        while((old_ename[x]=ename[x])!=' ')
                                ++x;
                        x = 0;
                        while((ename[x]=input[x])!=' ')
                                        ++x;
                        accept = redundancy_check(ename);
                        if (accept == 1)
                        {
                           finished = 1;
                           x = 0;
                           while((ename[x]=ole_ename[x])
                           !=' ')
                                ++x;
                           printf("0);
                           printf("Do you want to keep ");
                           printf("this element the way ");
                           printf("(y/n)0);
                           printf("(No will delete the ");
                           printf("element)0);
                           why = yes_no();
                           if (why == 1)
                           remove(ename);
                        }
                        if (finished == 0)
                        {
##                         replace e (name = ename) where
##                         e.description = edescript
                           for (x = 1; x <= count; x++)
                           {
##                              replace s (name = ename) where
##                              s.name = old_ename
                           }
                           what = 0;
##                         retrieve (eentity = i.entity)
##                         {
                                what++;
##                         }
                           for (x = 1; x <= what; x++)
                           {
##                              replace i (name = ename) where
##                              i.name = old_ename
                           }
                        }
                        break;
                   case '2':
                        x = 0;
                        while((edescript[x]=input[x])!=' ')
                                ++x;
##                      replace e (description = edescript)
##                      where e.name = ename
                        break;
                   case '3':
                        again = 0;
```

```
                    while (again == 0)
                    {
                      if((compare_strings(input,
                      "Character",9)==1)&&
                      (compare_strings(input,
                      "Alphanumeric",12)==1)&&
                      (compare_strings(input,"Integer",
                      7)==1)&&(compare_strings(input,
                      "Floating",8)==1))
                      {
                           printf("0he type must ");
                           printf("be either:0);
                           printf("    Character0);
                           printf("    Alphanumeric0);
                           printf("    Integer0);
                           printf("    Floating0);
                           printf("0ther input will ");
                           printf("not be ");
                           printf("accepted.0);
                           printf("0);
                           printf("New TYPE =   ");
                           get_input();
                      }
                      else
                           again = 1;
                    }
                    x = 0;
                    while((etype[x]=input[x])!=' ')
                           ++x;
##                  replace e (type = etype) where
##                  e.name = ename
                    break;
               case '4':
                    x = 0;
                    while((elength[x]=input[x])!=' ')
                           ++x;
##                  replace e (length = elength) where
##                  e.name = ename
                    break;
               case '5':
                    x = 0;
                    while((eentity_name[x]=input[x])
                    !=' ')
                           ++x;
##                  replace e (entity_name = eentity_name)
##                  where e.name = ename
                    break;
               case '6':
                    finished = 0;
                    x = 0;
                    while((ealias[x]=input[x])!=' ')
                           ++x;
                    accept = redundancy_check(ealias);
                    if (accept == 1)
```

```
                        {
                           finished = 1;
                           printf("0);
                           printf("Do you want to keep ");
                           printf("this element the way it");
                           printf(" was?(y/n)0);
                           printf("(No will delete the ");
                           printf("element)0);
                           why = yes_no();
                           if (why == 1)
                           remove(ename);
                        }
                        if (finished == 0)
                        {
##                         replace s (alias = ealias) where
##                         s.alias = old_alias
                        }
                        break;
                   case '7':
                        finished = 0;
                        printf("0);
                        printf("Which did you want to do?0);
                        printf("1) ADD0);
                        printf("2) DELETE0);
                        get_input();
                        if (input[0] =='1')
                        {
                           printf("What is the new alias?0);
                           get_input();
                           x = 0;
                           while((ealias[x]=input[x])!=' ')
                                ++x;
                           accept = redundancy_check(ealias);
                           if (accept == 1)
                           {
                                finished = 1;
                                printf("0);
                                printf("Do you want to keep");
                                printf(" this element the ");
                                printf("was it was?(y/n)0);
                                printf("(No will delete ");
                                printf("the alement)0);
                                why = yes_no();
                                if (why == 1)
                                remove(ename);
                           }
                           if (finished = 0)
                           {
##                              append to synonym (name =
##                              ename, alias = ealias)
                           }
                        }
                        else
                        {
```

```
                        printf("Which alias do you want");
                        printf(" to delete?0);
                        get_input();
                        x = 0;
                        while((ealias[x]=input[x])!=' ')
                             x++;
##                      delete s where s.alias = ealias
                     }
                     break;
                case '8':
                     printf("Which do you want to ");
                     printf("change?0);
                     printf("1) KEYWORD0);
                     printf("2) ENTITY0);
                     get_input();
                     number = input[0];
                     printf("What do you want to ");
                     printf("change it to?0);
                     get_input();
                     if (number == '1')
                     {
                        x = 0;
                        while((ekeyword[x]=input[x])
                        !=' ')
                             ++x;
##                      replace i (keyword = ekeyword)
##                      where i.entity = old_eentity
                     }
                     else
                     {
                        x = 0;
                        while((eentity[x]=input[x])!=' ')
                               ++x;
##                      replace i (entity = eentity) where
##                      i.entity = old_eentity
                     }
                     break;
                case '9':
                     printf("Which do you want to do?0);
                     printf("1)  ADD0);
                     printf("2)  DELETE0);
                     get_input();
                     if (input[0] == '1')
                     {
                        printf("Data Element KEYWORD =  ");
                        get_input();
                        x = 0;
                        while ((ekeyword[x]=input[x])!=' ')
                             x++;
                        printf("Data Element ENTITY =  ");
                        get_input();
                        x = 0;
                        while ((eentity[x]=input[x])!=' ')
                             x++;
```

```
##                               append to instance (name = ename,
##                               keyword = ekeyword, entity =
##                               eentity)
                             }
                             else
                             {
                                printf("Which instance gets ");
                                printf("zapped?0);
                                printf("(Signify be entity)0);
                                get_input();
                                x = 0;
                                while((eentity[x]=input[x])!=' ')
                                     x++;
##                              delete i where i.entity = eentity
                             }
                             break;
                        default:
                             printf("Not an option0);
                             loop = 0;
                   }
        }
        count = print_element(ename);
}

/*************************************************************/
/**************                         *********************/
/**************      DELETE_ELEMENT     *********************/
/**************                         *********************/
/*************************************************************/

delete_element()
{
        extern char input[];
        int x, back;

        printf("Do you want to see a list of ");
        printf("tbe elements?(y/n)0);
        back = yes_no();
        if (back == 0)
        {
                short_list();
        }
        printf("What is the name of the element ");
        printf("that you want to delete?0);
        get_input();
        x = 0;
        while ((ename[x] = input[x]) != ' ')
                x++;
        printf("Are you sure that you want to delete ");
        printf("tbe %s element?0,ename);
        back = yes_no();
        if (back == 0)
                remove(ename);
        short_list();
```

```
}
/*************************************************************/
/*************                         ********************/
/*************       SHORT_LIST        ********************/
/*************                         ********************/
/*************************************************************/

short_list()
{
        printf("0);
        printf("Here are the elements in the ");
        printf("dictionary now.0);
##      retrieve (ename = e.name)
##      {
                printf("%s0, ename);
##      }
}


/*************************************************************/
/*************                         ********************/
/*************         REMOVE          ********************/
/*************                         ********************/
/*************************************************************/

remove(word)

{
##      delete e where e.name = word
##      delete s where s.name = word
##      delete i where i.name = word
}

/*************************************************************/
/*************                         ********************/
/*************    REDUNDANCY_CHECK     ********************/
/*************                         ********************/
/*************************************************************/

redundancy_check(word)

/* The purpose of this function is to take a data element */
/* name or alias, and check it against the rest of the     */
/* data dictionary for any kind of redundancies.           */

{
        int found;

        found = 0;
##      retrieve (fill = e.name) where e.name = word
##      {
                found = 1;
##      }
        if (found != 1)
```

```
        {
##              retrieve (fill = s.alias)
##              where s.alias = word
##              {
                        found = 1;
##              }
        }
        if (found == 1)
        {
                printf("0);
                printf("0);
                printf("ERROR---A REDUNDANT DATA ELEMENT ");
                printf("HAS BEEN INPUT!0);
                printf("IT IS BEING REMOVED FROM THE ");
                printf("DATA DICTIONARY!0);
        }
        return(found);
}

/**************************************************************/
/**************                         *********************/
/**************         CREATE_DB       *********************/
/**************                         *********************/
/**************************************************************/

create_db()
/* This function does the actual creation of the */
/* data base schema. */

{
        int answer, done, x, y, z, w, number;
        int times[max_elements];
##      char entities[max_elements][maxsize];
##      struct buffer{
##              char entity[maxsize];
##              char attribute[maxsize];
##              char keyword[2];
##              char type[5];
##      } entity_buffer[max_elements];

        printf("Before the data dictionary can create ");
        printf("a data base schema 0);
        printf("from the data definitions that you have ");
        printf("input, you have 0);
        printf("needed to have created a data base in ");
        printf("INGRES.  This is done0);
        printf("with the following command (if you have ");
        printf("the capability to 0);
        printf("create a data base)0);
        printf("                creatdb <data base name>0);
        printf("0);
        printf("Have you already created a data base ");
        printf("in INGRES?(y/n)0);
        answer = yes_no();
```

```
        if (answer == 0)
        {
           printf("0);
           printf("What is the name of the data base ");
           printf("to be created?0);
           get_input();
           x = 0;
           while((dict_name[x]=input[x])!=' ')
                x++;
           number = 0;
           y = 0;

           /* First find all of the entity names */

##         retrieve (ename = e.name) where e.entity_name = "Y"
##         {
                x = 0;
                while((entities[y][x]=ename[x])!=' ')
                      x++;
                y++;
                number++;
##         }

           /* Find all of the attributes for the entity   */
           /* names and whether they are keywords.        */

           z = 0;
           for (w = 0; w < number; w++)
           {
                times[w] = 0;
##              retrieve (ename = i.name, ekeyword =
##              i.keyword) where i.entity = entities[w]
##              {
                   times[w]++;
                   x = 0;
                   while((entity_buffer[z].entity[x]=
                   entities[w][x])!=' ')
                        x++;
                   x = 0;
                   while((entity_buffer[z].attribute[x]=
                   ename[x])!=' ')
                        x++;
                   x = 0;
                   while((entity_buffer[z].keyword[x]=
                   ekeyword[x])!=' ')
                        x++;
                   z++;
##              }
           }

           /* Find the types and lengths of each of the */
           /* attributes. */

           for (y = 0; y < z; y++)
```

```
            {
##              retrieve (etype = e.type, elength = e.length)
##              where e.name = entity_buffer[y].attribute
##              {
                   if((compare_strings(etype,"Character",9)==
                   0)||(compare_strings(etype,"Alphanumeric"
                   ,12)==0))
                        entity_buffer[y].type[0]='c';
                   if(compare_strings(etype,"Integer",7)==0)
                        entity_buffer[y].type[0]='i';
                   if (compare_strings(etype,"Floating",8)==0)
                        entity_buffer[y].type[0]='f';
                   x = 0;
                   while((entity_buffer[y].type[x+1]=
                   elength[x])!=' ')
                        x++;
##              }
            }

            /* Do the actual creation of the data base schema*/

##          exit
##          ingres dict_name

            z = 0;
            for (x = 0; x < number; x ++)
            {
                 switch (times[x])
                 {
                        case 1:
##                         create entity_buffer[z].entity(
##                         entity_buffer[z].attribute=
##                         entity_buffer[z].type)
                           break;
                        case 2:
##                         create entity_buffer[z].entity(
##                         entity_buffer[z].attribute=
##                         entity_buffer[z].type,
##                         entity_buffer[z+1].attribute=
##                         entity_buffer[z+1].type)
                           break;
                        case 3:
##                         create entity_buffer[z].entity(
##                         entity_buffer[z].attribute=
##                         entity_buffer[z].type,
##                         entity_buffer[z+1].attribute=
##                         entity_buffer[z+1].type,
##                         entity_buffer[z+2].attribute=
##                         entity_buffer[z+2].type)
                           break;
                        case 4:
##                         create entity_buffer[z].entity(
##                         entity_buffer[z].attribute=
##                         entity_buffer[z].type,
```

```
##                              entity_buffer[z+1].attribute=
##                              entity_buffer[z+1].type,
##                              entity_buffer[z+2].attribute=
##                              entity_buffer[z+2].type,
##                              entity_buffer[z+3].attribute=
##                              entity_buffer[z+3].type)
                                break;
                            case 5:
##                              create entity_buffer[z].entity(
##                              entity_buffer[z].attribute=
##                              entity_buffer[z].type,
##                              entity_buffer[z+1].attribute=
##                              entity_buffer[z+1].type,
##                              entity_buffer[z+2].attribute=
##                              entity_buffer[z+2].type,
##                              entity_buffer[z+3].attribute=
##                              entity_buffer[z+3].type,
##                              entity_buffer[z+4].attribute=
##                              entity_buffer[z+4].type)
                                break;
                            case 6:
##                              create entity_buffer[z].entity(
##                              entity_buffer[z].attribute=
##                              entity_buffer[z].type,
##                              entity_buffer[z+1].attribute=
##                              entity_buffer[z+1].type,
##                              entity_buffer[z+2].attribute=
##                              entity_buffer[z+2].type,
##                              entity_buffer[z+3].attribute=
##                              entity_buffer[z+3].type,
##                              entity_buffer[z+4].attribute=
##                              entity_buffer[z+4].type,
##                              entity_buffer[z+5].attribute=
##                              entity_buffer[z+5].type)
                                break;
                            case 7:
##                              create entity_buffer[z].entity(
##                              entity_buffer[z].attribute=
##                              entity_buffer[z].type,
##                              entity_buffer[z+1].attribute=
##                              entity_buffer[z+1].type,
##                              entity_buffer[z+2].attribute=
##                              entity_buffer[z+2].type,
##                              entity_buffer[z+3].attribute=
##                              entity_buffer[z+3].type,
##                              entity_buffer[z+4].attribute=
##                              entity_buffer[z+4].type,
##                              entity_buffer[z+5].attribute=
##                              entity_buffer[z+5].type,
##                              entity_buffer[z+6].attribute=
##                              entity_buffer[z+6].type)
                                break;
                    }
##                  print entity_buffer[z].entity
```

```
                z = z + times[x];
            }
        }
}

/************************************************************/
/*************                          *******************/
/*************    COMPARE_STRINGS       *******************/
/*************                          *******************/
/************************************************************/

compare_strings(string1, string2, size)
char string1[maxsize], string2[maxsize];
int size;

{
        int x, same;

/* If the two strings are the same, */
/* return a 0, otherwise a 1 */
        same = 0;
        for (x = 0; x < size; x++)
        {
                if(string1[x]!=string2[x])
                        same = 1;
        }
        return(same);
}
```

A DATA DICTIONARY FOR THE INGRES
DATA BASE MANAGEMENT SYSTEM

by

LOREN WILSON

B. A., Kansas Wesleyan, 1980

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

A data dictionary is a data base software tool that many organizations are using to help them in the control of their data resources. It is primarily used to hold "what" a data element is, and "where" it can be found. This information is known as metadata.

The first thing this paper does is to define what the functions of a normal data dictionary are. Then it differentiates between different types of data dictionaries: static, dynamic, DBMS-dependent, and stand-alone.

The primary thrust of this paper is to develop a data dictionary that can be used in the design process of a data base to store all of the data definitions of the data base. Then the data dictionary dynamically creates a relational data base schema from these data definitions.

The data dictionary is defined starting with the data dictionary data base, the input programs, and the output programs. Then the process that the data dictionary uses to dynamically create the data base is discussed. An example data base is created going through all of the steps outlined.